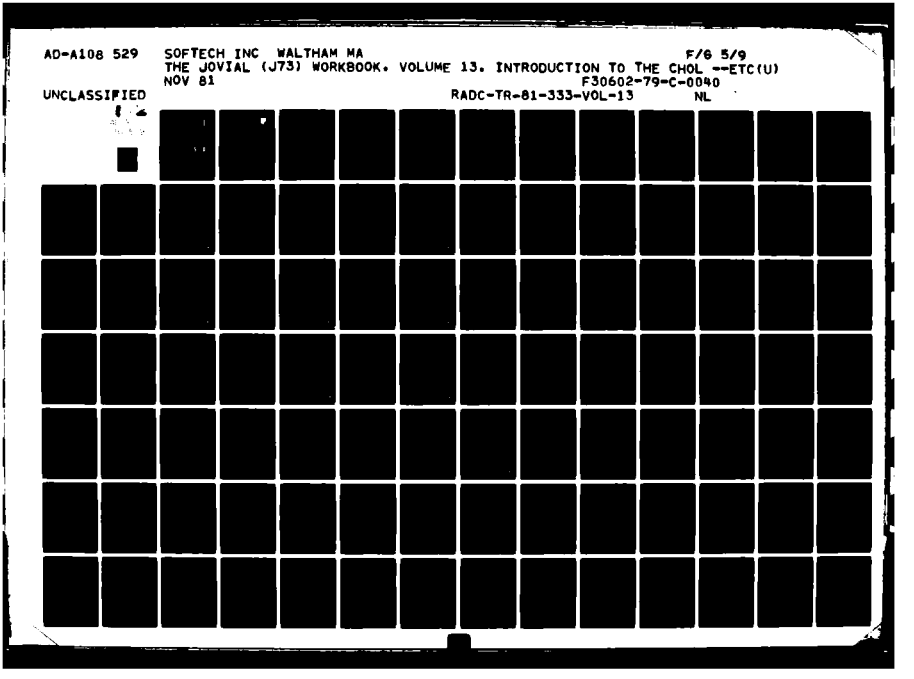
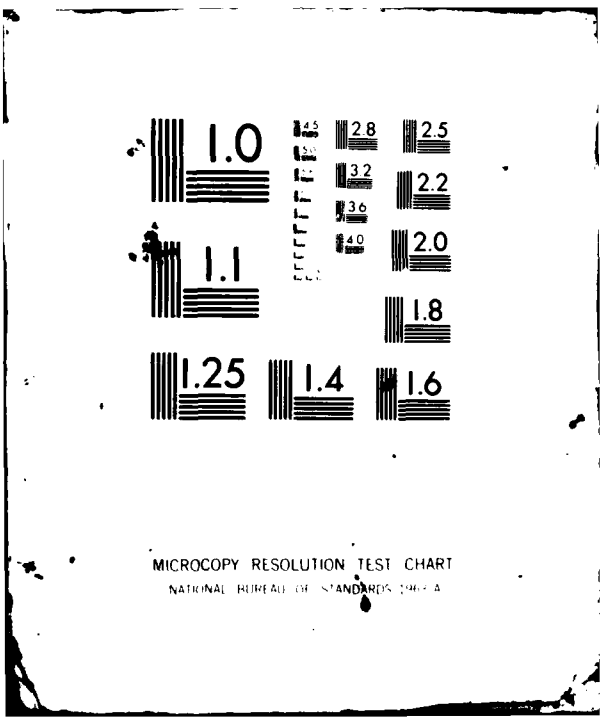
AD-A108 529 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME 13. INTRODUCTION TO THE CHOL --ETC(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-13 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

PHOTOGRAPH THIS SHEET

AD A108529

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Softech, Inc.
Waltham, MA

The JOVIAL (J73) Workbook
Nov. 81

DOCUMENT IDENTIFICATION

Contract F30602-79-C-0040 RADC-TR-81-333, Vols. XIII-XV (of 15)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS	GRA&I
DTIC	TAB
UNANNOUNCED	
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A	

DISTRIBUTION STAMP

DTIC
ELECTE
S DEC 14 1981 D
D

DATE ACCESSIONED

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-333, Vols XIII-XV (of 15)
Interim Report
November 1981

THE JOVIAL (J73) WORKBOOK

SofTech, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-333, Vols XIII-XV (of 15) have been reviewed and are approved for publication.

APPROVED: Douglas A. White

DOUGLAS A. WHITE
Project Engineer

APPROVED: John J. Marciniak

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-333, Vols XIII - XV (of 15)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE JOVIAL (J73) WORKBOOK		5. TYPE OF REPORT & PERIOD COVERED Interim Report Dec 79 - Oct 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) N/A		8. CONTRACT OR GRANT NUMBER(s) F30602-79-C-0040
9. PERFORMING ORGANIZATION NAME AND ADDRESS SofTech, Inc. 460 Totten Pond Rd Waltham MA 02154		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 33126F 20220403
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE November 1981 13. NUMBER OF PAGES 113
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Douglas A. White (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

JOVIAL (J73)
MIL-STD-1589A
Video Course
Higher Order Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The JOVIAL (J73) Workbook is only one portion of a self-instructional JOVIAL (J73) training course. In addition to the programmed-learning primer/workbooks, are video taped lectures. The workbooks are formatted to consist of fifteen (15) segments bound in three (3) volumes covering each particular language capability. A video tape lectrue was prepared for each workbook segment. This course is taught in two parts. Part I contains twelve (12) segments in Volumes I and II of the workbook; Part

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

II, Volume III contains three (3) segments. There is a brief explanatory introduction at the beginning of the course. Each of the individual segments deals with a specific feature of the JOVIAL language. The video tapes act as an overview to outline particular points that are followed up in the written workbooks. Each tape runs a maximum of 25 minutes and contains an average of 15 graphic each.

THE JOVIAL (J73) WORKBOOK

VOLUME 13

INTRODUCTION TO THE CHOL EXTENSIONS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

Workbook 13 is intended for use with Tape 13 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

The JOVIAL (J73/C) programming language takes the (JOVIAL (J73) language and extends it with features designed primarily for communications programming. These extensions are introduced in this workbook. Specific attention is given to items, statements, table declarations, subroutines, built-in functions and directives. Exercises are also provided where appropriate.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction- a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' *indicates one or more* repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

ITEMS

ITEMS

An item is a scalar variable or constant. All items must be declared before they can be used anywhere in a program. The general syntax for an item-declaration in JOVIAL (J73) is:

[CONSTANT] ITEM name { type-description / item-type-name } [item-preset];

The type-descriptions shown so far are:

U [integer-size]

S [integer-size]

F [precision]

A scale [,fraction]

B [bit-size]

C [character-size]

STATUS [size] ([status-index] V ({ name / letter / reserved-word }), ...)

P [type-name]

An item-declaration specifies that an item-name designates a variable or a constant with a specified type-class and attributes. (Workbooks 1, 2 and 5 should be reviewed for details.)

This section discusses the value range specification available in JOVIAL (J73/C), as well as the ALPHA status type, character formulae, subroutine items and some changes in type matching rules.

VALUE RANGE SPECIFICATION

The value range specification restricts the values which an item may assume. A value-range may be given as part of the type-description for **numeric types**. Value-range is of the form:

[,] *lower-bound*: *upper-bound*

Lower-bound and upper-bound are formulae of the appropriate type, and must be known at compile-time. Lower bound must be less than or equal to upper bound. The comma before the value range is used if and only if both a size attribute and a value-range are specified.

The errors signalled are as follows:

- Value of item-preset out of range of value
- Value of source in an assignment out of value-range of target
- Value of formulae out of range of conversion operator or scale
- Value of source in an assignment out of range implied by integer-size or scale

Value Range Specification: Examples

Declaration	Meaning
ITEM INDEX U 5, 1:25;	INDEX designates an unsigned 5-bit integer variable that can take on values in the range 1 through 25.
ITEM TOLERANCE S -1000:1000;	TOLERANCE designates a signed integer variable that can assume values in the range -1000 through +1000. It will be allocated at least ten bits for the signed integer value.
ITEM ERROR F, .00001:.001;	ERROR designates a floating variable with default precision. It can assume values in the range .00001 through .001.
ITEM SPEED A 10, 3, 1.:1000.;	SPEED designates a fixed variable with scale 10 and fraction 3 that can assume values in the range 1 through 1000.
ITEM THICKNESS A 7, 5.5:102.125;	THICKNESS designates a fixed variable with scale 7 and fraction FIXED-PRECISION - 7 that can assume values in the range 5.5 through 102.125.

ALPHA STATUS TYPE

As mentioned in Workbook 2, a status item is an item whose value range is a specified list of symbolic names called status-constants. A status-constant is a symbolic constant that has an ordering relative to the other status-constants in the list. A status item provides a way enumerate all possible values for that item in a mnemonic way.

In JOVIAL (J73/C) there is a built-in status type called ALPHA that corresponds to the bit patterns represented by the character set of a machine. The ALPHA status-constants are represented in the following way:

V(NUL), V(BEL)	non-printing character
V($A), V($J)	lower-case character
V(A), V(J)	upper-case character
V(1), V(9)	numerals with character-set bit representation

ALPHA has default representation. A complete list of ALPHA status constants can be found in Figure 1-1.

Status-Constant	Value in Octal	Description	Status-Constant	Value in Octal	Description
V(NUL)	0	null	V(DC3)	23	device control 3
V(SOM)	1	start of message	V(DC4)	24	device control 4
V(SOT)	2	start of text	V(NACK)	25	negative acknowledge
V(EOT)	3	end of text	V(IDL)	26	synchronous idle
V(EOTR)	4	end of trans.	V(EOB)	27	end of block
V(ENQ)	5	enquiry	V(CAN)	30	cancel
V(ACK)	6	acknowledge	V(EOMD)	31	end of medium
V(BEL)	7	bell	V(SUB)	32	substitute
V(BS)	10	backspace	V(ESC)	33	escape
V(HT)	11	horizontal tab	V(FS)	34	file separator
V(LF)	12	line feed	V(GS)	35	group separator
V(VT)	13	vertical tab	V(RS)	36	record separator
V(FF)	14	form feed	V(US)	37	unit separator
V(CR)	15	carriage return	V(BL)	40	blank
V(SO)	16	shift out	V(DEL)	177	delete
V(SI)	17	shift in	V($A) to V($Z)	141-172	lowercase alphabet
V(DLE)	20	data link escape	V(prntg char)	ASCII code	others
V(DC1)	21	device control 1			
(VDC2)	22	device control 2			

Figure 1-1. Status Constants in ALPHA

CHARACTER FORMULAE

A character formula represents a value whose type class is character. Its size is the number of bytes comprising its value. In JOVIAL (J73/C), a character formula known at compile-time may use the concatenation operator (11). The type of the result is Cn where n is the sum of the sizes of the character strings.

Example

```
ITEM NAME C9 = 'JOHN'//'SMITH';
```

The ALPHA status type discussed above is very useful in this context, as in the following example:

```
ITEM NAME C 10 = 'JOHN'//V(CR)//'SMITH';
```

The status-constant V(CR) - carriage return - is implicitly converted to a C 1 in this example.

SUBROUTINE ITEMS

A subroutine item is used to indicate a function or procedure. A subroutine item may be assigned to reference a specific function or procedure. The subroutine item-name may then be used to invoke the subroutine that is its current value.

The form of a subroutine type-description is:

```
PROC [ use-attribute ] [ ( formal-list ) ] [ type-description ] ;
        parameter-declaration
```

The square brackets indicate that use-attribute, the parenthesized formal-list and type-description are all optional. If a subroutine type-description contains a type-description, the name being declared designates a function item. Otherwise, it designates a procedure item.

Parameter-declaration is a declaration. If only one parameter is used, the parameter-declaration is a simple declaration. If more than one parameter is used, the parameter-declaration is compound. If no parameters are used, the parameter-declaration must be a null-declaration.

Some examples of subroutine item-declarations are:

Declaration	Meaning
ITEM FLOOR PROC (ARG) S; ITEM ARG S;;	FLOOR designates a function item with one argument, which is a signed integer. The return value of FLOOR is a signed integer. If it is declared within a subroutine, it has static allocation. Otherwise, it has automatic allocation. Its initial value is unspecified.
ITEM CLEAR PROC;;;	CLEAR is a procedure item with no parameters. Its initial value is unspecified.
ITEM MULT PROC (M1, MS); BEGIN TABLE M1 (100); ITEM MUL1 U; TABLE M2 (100); ITEM MUL2 U; END	MULT is a procedure item with two table parameters.

A subroutine item may be used almost anywhere a simple item may be used. However, a subroutine item must not be used as an input or output parameter or as the type of a function return value.

SUBROUTINE VARIABLES

A subroutine-item may receive an assignment and may be invoked.

Example

```
ITEM SORT PROC(: LIST);
      TABLE LIST(*);
            ITEM VALUE U;
      ...
SORT = FASTSORT;
SORT (: DATA);
```

```
...
PROC FASTSORT(: LIST);
        subroutine-definition
PROC SLOWSORT(: LIST);
        subroutine-definition
```

In this example, item SORT is of type subroutine -- procedure in this case The procedure-call-statement invokes the procedure FASTSORT.

SUBSCRIPTED SUBROUTINE VARIABLES

A subroutine item, like any other item may be subscripted. The form is:

subroutine-item-name (subscript-list) (actual-list)

The subscripted subroutine item may be used almost anywhere a simple item may be used.

Example

```
TABLE EMPLOYEES(1 : NUM'EMP);
        BEGIN
        ITEM NAME C 20;
        ITEM SOC'SEC'NO C 9;
        ITEM PAY'KIND STATUS (V(SALARIED), V(HOURLY));
        ITEM PAYCHECK F;
        ITEM COMPUTERPAY PROC (WORK'HOURS'THIS'MONTH) F;
                ITEM WORK'HOURS'THIS'MONTH U;
        END
```

```
...
WORK'HOURS'THIS'MONTH = 160;
FOR I : 1 TO NUM'EMP;
        BEGIN
        IF PAY'KIND(I) = V(SALARIED);
                COMPUTEPAY(I) = SALARY'PLUS'BONUS;
        ELSE
        COMPUTEPAY(I) = WAGES'PLUS'OVERTIME;
        PAYCHECK(I) = COMPUTEPAY(I)(WORK'HOURS'THIS'MONTH);
        END
...
PROC SALARY'PLUS'BONUS(HOURS) F;
        subroutine-body
PROC WAGES'PLUS'OVERTIME(HOURS)F;
        subroutine-body
```

CONSTANT AND TYPED SUBROUTINE ITEMS

A subroutine item may be declared to be a constant or may be declared with an item-type-name.

Examples

```
CONSTANT ITEM SALARY PROC(EMPTAB) F;
        TABLE EMPTAB;
                BEGIN
                ITEM PERSON C 20;
                ITEM PAY F;
                END = PAY'WEEK(EMPTAB);
```

```
TYPE WAGES PROC(EMPTAB) F;
    TABLE EMPTAB;
        BEGIN
        ITEM PERSON C 20;
        ITEM PAY F;
        END
ITEM PAYCHECK WAGES;
ITEM EXTRAPAY WAGES = PAY'SUN(EMPTAB);
```

(Both PAY'WEEK and PAY'SUN both have their function-definitions somewhere else in the program.)

PRESETS

Like all other items, a subroutine item may be preset. The form is:

ITEM name type-declaration item-preset;

Examples

```
ITEM SUBR1 PROC(IN); ITEM IN U ; = FIGURE1(WAGES);
```

The first example shows SUBR1, a subroutine item, with one parameter, IN, its declaration, and an item-preset.

```
ITEM SUBR2 PROC ; ; = FIGURE2;
```

The second example shows SUBR2, a subroutine item, with no parameters but with an item-preset. (A null parameter-declaration must be given.)

TYPE MATCHING AND CONVERSION

Because of the addition of value range specifications and subroutine items, there are some additional rules for type matching and conversion in J73/C. These are summarized below.

INTEGER TYPES (SIGNED AND UNSIGNED)

Type Equivalence:	Two integer types are equivalent if they are both S or U and if their (explicit or default) size attributes and **subrange attributes are equal.**
Implicit Conversions:	An integer type will be implicitly converted to any other integer type, **provided their subranges are not disjoint.**
Explicit Conversions:	Unchanged.

FLOATING POINT TYPES

Type Equivalence:	Two floating types are equivalent if their (explicit or default) precision attributes and **subrange attributes are equal.**
Implicit Conversions:	A floating type will be implicitly converted to a floating type of the same or greater precision regardless of the round-or-truncate attribute, **provided their subranges are not disjoint.**
Explicit Conversions:	Unchanged.

FIXED POINT TYPES

Type Equivalence:	Two fixed point types are equivalent if their scale attributes are equal and their (explicit or default) fraction attributes **and subrange attributes are equal.**

Implicit Conversions: A fixed point type will be implicitly converted to another fixed point type if the scale and fraction attributes of the target type are both at least as large of those of the source type **and if their subranges are not disjoint.**

Explicit Conversions: Unchanged.

STATUS TYPES

Conversion rules unchanged.

BIT TYPES

Conversion rules unchanged.

CHARACTER TYPES

Conversion rules unchanged.

SUBROUTINE TYPE

Type Equivalence: **Two subroutine types are equivalent if they are declared with the same subroutine-attributes, their parameters agree in number, order, type, and binding, and their result-types (if any) are equivalent. Note that the names of the parameters do not have to be the same.**

Implicit Conversions: **No implicit conversions are performed.**

Explicit Conversions: **No explicit conversions are performed.**

SECTION 2

EXECUTABLE STATEMENTS

EXECUTABLE STATEMENTS

Executable statements in JOVIAL (J73) were discussed in Workbook 3. JOVIAL (J73/C) provides extended features for loop control, loop exiting, the deallocation of heap storage, and the protection of data shared by concurrent processes.

LOOP STATEMENTS: THE TO-PHRASE

J73/C allows the programmer to specify a to-clause to further control the execution of a loop. The general syntax is:

```
FOR loop-control : initial-value [BY increment]
  [TO limit] [WHILE condition];
```

A to-clause (or a while-phrase) controls the number of times the statement in the for-loop is to be executed. In a to-clause, limit is a numeric or status formula. If limit is a floating or fixed formula, a by-clause must be given. If limit is an integer formula and a by-clause is not present, loop-control is incremented by 1. If limit is a status formula, loop-control is incremented by taking the successor of the current status value. (In a while-phrase, condition is a Boolean formula.)

The execution of the incremented for-loop is as follows:

1. Evaluate initial-value and assign its value to loop-control. If a to-clause is given, evaluate limit and increment (if present).

2. If a to-clause is not given, continue to step 3. If a to-clause is given, compare the value of loop-control with the value of limit. Execution continues if:

   loop-control < limit and increment is positive.

   loop-control > limit and increment is negative.

3. Evaluate condition if the while-clause (if present). If the value of condition is TRUE, continue to step 4. If the value of the condition is FALSE, terminate the for-loop.

4. Execute the controlled-statement.

5. Evaluate increment and add it to loop-control. Return to step 2.

EXIT STATEMENT: THE EXIT LABEL

An exit-statement is used to exit a loop at the point within the loop where the exit-statement is executed.

An exit-statement exits from the immediately enclosing loop if no exit-label is specified in the exit-statement, or from the loop labelled by the exit-label in the exit statement. The form of the exit-statement is:

EXIT [exit-label] ;

The square brackets indicate that exit-label is optional.

Exit-label must be a statement-name used as a label on an enclosing loop. If exit-label is given, the exit is from the loop labelled with the statement-name. If exit-label is not given, the exit is from the immediately enclosing loop-statement.

The effect of an exit-statement is the same as the effect of a goto-statement that transfers control out of the controlled-statement.

Example

The following for-loop may be written to sum the items of a table with two dimensions and to terminate the summation process if the sum exceeds a specified threshold. Since the table has two dimensions, the summation process requires a nested for-loop. A labelled exit-statement is used to terminate the execution of both for-loops:

```
OUTERLOOP: FOR I : 1 to 100;
        INNERLOOP: FOR J : 1 TO 100;
                BEGIN
                SUM = SUM + COUNT (I, J);
                IF SUM > IMAX;
                        EXIT OUTERLOOP;
```

```
        IF SUM > JMAX;
            EXIT;
        END
```

UPDATE STATEMENT

The update-statement is used to control access to data that is shared by concurrent processes. Data that is shared in this way must have PROTECTED allocation specified in its declaration and may only be accessed within an update-statement.

The form of an update-statement is:

```
UPDATE
        protection-list
        statement ...
        [condition-handler]
        [ label ... ] END
```

The protection-list is a sequence of one or more data names, separated by commas. It names the protected data that is to be referenced in the update-statement.

For each data name in the protection-list, a lock routine is called at the beginning of the update-statement and an unlock routine at the end of the update-statement. The locking of the data prevents its being accessed by another process.

Given the following table-declaration:

```
TABLE INTEREST PROTECTED (9);
        ITEM RATE F;
```

If one process periodically changes the interest rates and other processes use the interest rates, the process that changes the rates may use an update-statement, as follows:

```
UPDATE INTEREST
        FOR I : 0 TO 9;
                RATE (I) = FACTOR (1 + RATE (I));
END
```

The processes that use the interest rates may also use update-statements to access them. For example:

```
UPDATE INTEREST
        CASE PERIOD;
                BEGIN
                (20): MORTGAGERATE = RATE (1);
                (25): MORTGAGERATE = RATE (2);
                (30): MORTGAGERATE = RATE (3);
                END
END
```

In this way, a process is ensured that the data being accessed is internally consistent.

If an update-statement is to be executed when the data to be accessed is locked by another process, the execution waits until the data is unlocked. Careless use of the update-statement can result in a deadlocked program.

FREE PROCEDURE CALL STATEMENT

The FREE procedure returns storage referenced by a pointer-formula to the heap. The FREE procedure is called as follows:

```
FREE (pointer-formula);
```

The pointer-formula must be a typed pointer and must point to an object that was allocated by the NEW function (discussed below).

SECTION 3

TABLES

TABLES

JOVIAL (J73) table declarations were discussed in Workbook 4. J73/C provides three extensions to J73 - table nesting, case variants and the use of tables in relational expressions.

NESTED TABLES

A nested table-declaration is a table-declaration given as a table-option within another table-declaration.

The following table-declaration contains a nested table-declaration:

```
TABLE TEAM;
      BEGIN
      ITEM NAME C 15;
      ITEM CITY C 15;
      TABLE RECORD;
            BEGIN
            ITEM WINS U;
            ITEM LOSSES U;
            END
      ITEM OWNER C 15;
      END
```

This table may be diagrammed as follows:

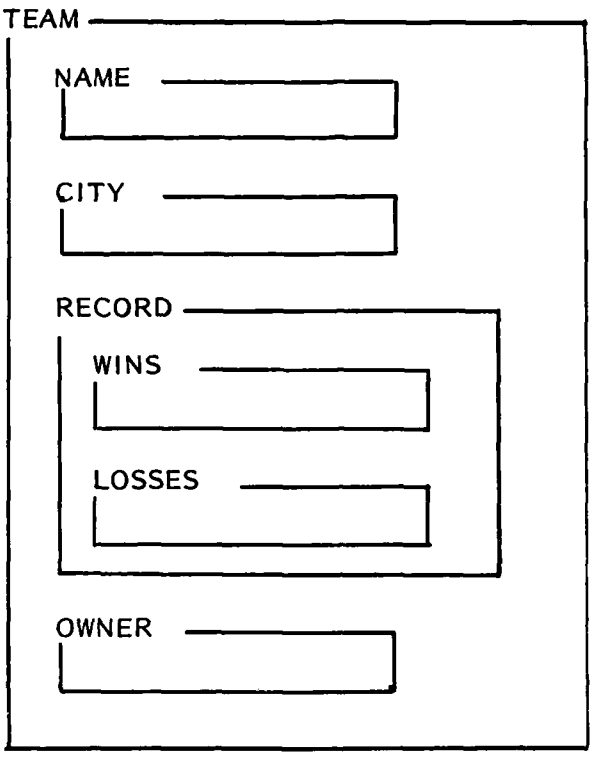


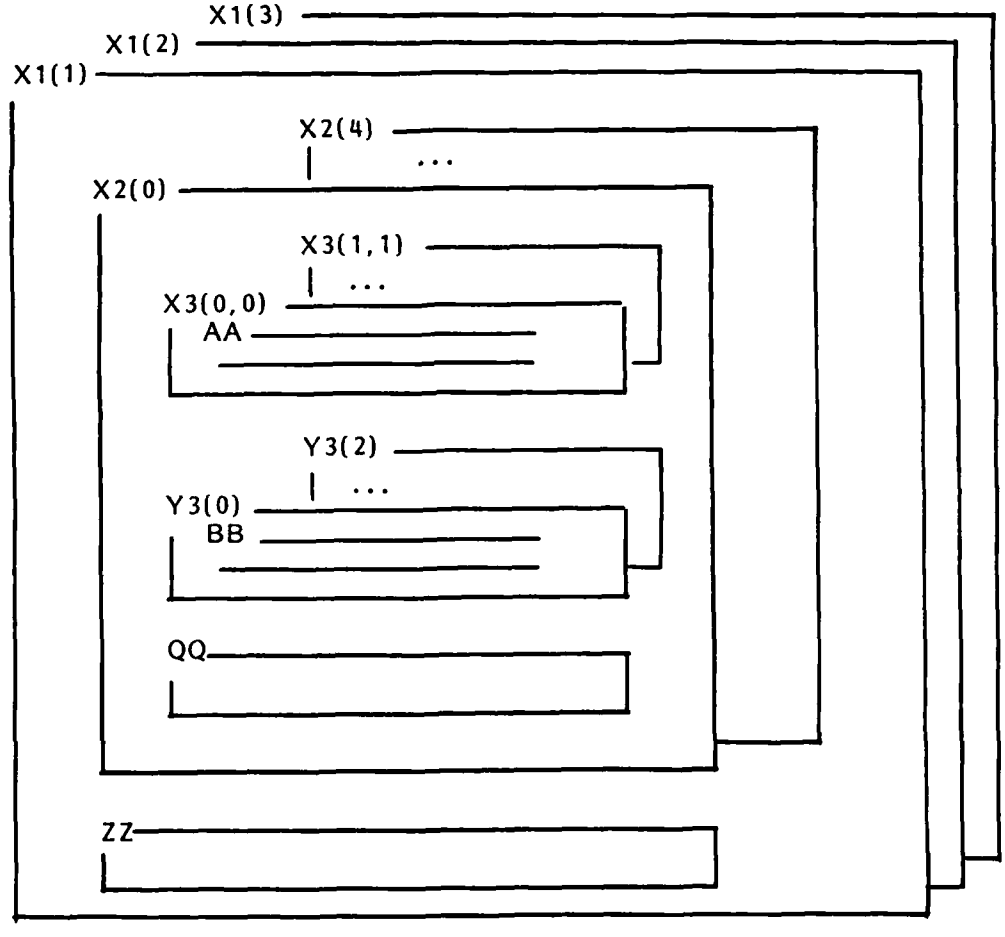
Table TEAM has one entry. This entry contains the item NAME, the item CITY, the table RECORD, and the item OWNER.

The following is another example of a nested table-declaration:

```
TABLE X1 (1 : 3);
        BEGIN
        TABLE X2 (4);
                BEGIN
                TABLE X3 (1, 1);
                        ITEM AA U;
```

```
        TABLE Y3 (2);
            ITEM BB U;
        ITEM QQ F;
        END
    ITEM ZZ B 12;
    END
```

The table X1 may be diagrammed as follows:



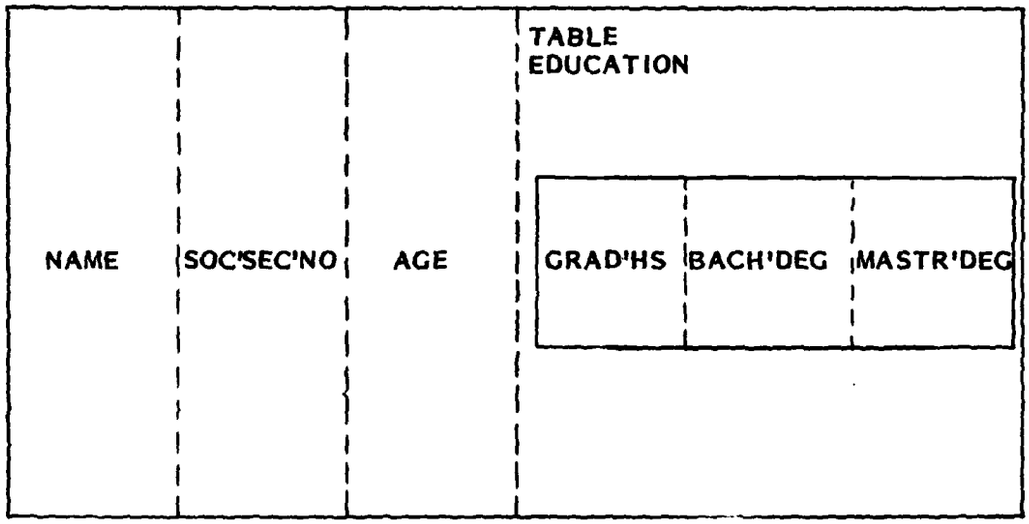
The table X1 has three entries. Each entry contains a nested table X2 and an item ZZ. The nested table X2 has five entries. Each entry contains a nested table X3, a nested table Y3, and an item QQ. The nested table X3 has two dimensions and four entries, each containing an item AA. The nested table Y3 has one dimension and three entries, each containing an item BB.

As a final example, consider table DETAIL'ID:

```
TABLE DETAIL'ID
        BEGIN
        ITEM NAME C 20;
        ITEM SOC'SEC'NO C 9;
        ITEM AGE U 7;
        TABLE EDUCATION;
                BEGIN
                ITEM GRAD'HS B;
                ITEM BACH'DEG B;
                ITEM MASTR'DEG B;
                END
        END
```

NOTES: The table DETAIL'ID has one entry. That entry is composed of three items (NAME, SOC'SEC'NO, and AGE) and one table (EDUCATION). The table called EDUCATION has one entry. That entry is composed of three items (GRAD'HS, BACH'DEG, and MASTR'DEG).

DETAIL'ID could be diagrammed as follows:



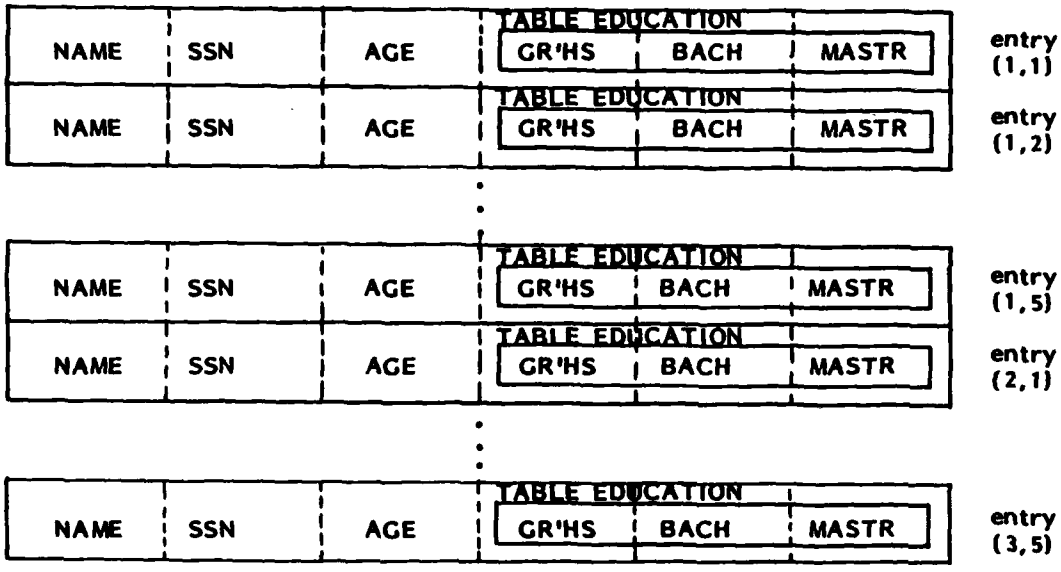
NESTED TABLES IN DIMENSIONED TABLES

A dimensioned table may include a nested table.

Example

```
TABLE DETAIL'ID (1 : 3, 1 : 5);
        BEGIN
        ITEM NAME C 20;
        ITEM SOC'SEC'NO C 9;
        ITEM AGE U 7;
        TABLE EDUCATION
                BEGIN
                ITEM GRAD'HS B;
                ITEM BACH'DEG B;
                ITEM MASTER'DEG B;
                END
        END
```

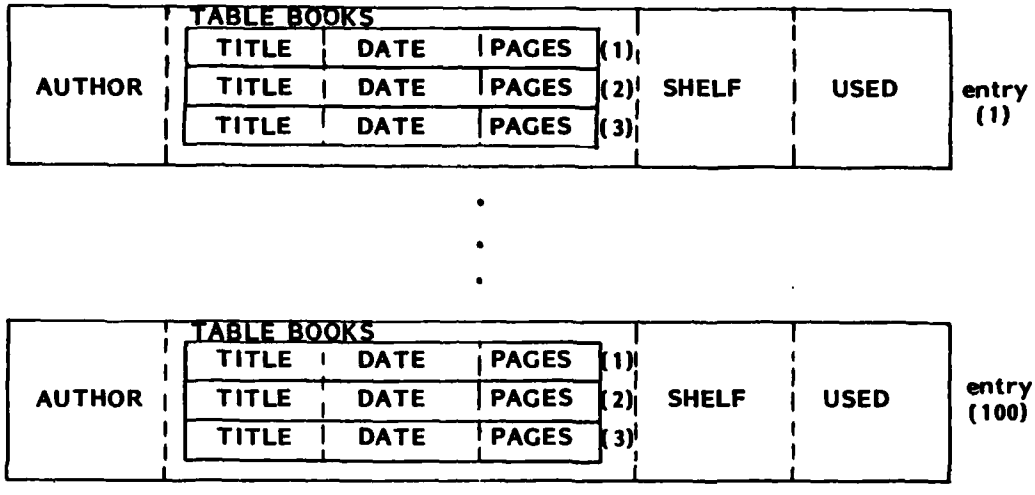
The table DETAIL'ID is a two-dimensional table, each entry has three items and a table with three items. DETAIL'ID may be diagrammed as follows:



As another example, consider table LIBRARY:

```
TABLE LIBRARY(1 : 100);
        BEGIN
        ITEM AUTHOR C 20;
        TABLE BOOKS(1 : 3);
                BEGIN
                ITEM TITLE C 10;
                ITEM DATE C 8;
                ITEM PAGES U;
                END
        ITEM SHELF C 3;
        ITEM USED B
        END
```

The table LIBRARY is a one-dimensional table, each entry has three items and another one-dimensional table with three items. LIBRARY can be diagrammed as follows:



NESTED TABLE PRESETS

A table-preset may appear on a table-heading provided it does not preset any items in a nested table. If any items in a nested table are to be preset, the preset must go on the nested table or the items within the nested table. The following examples preset the items NAME, GRAD'HS, and BACH'DEG.

```
TABLE DETAIL'ID
        BEGIN
        ITEM NAME C 20 = 'J SMITH';
        ITEM SOC'SEC'NO C 9;
        ITEM AGE U 7;
        TABLE EDUCATION;
```

```
            BEGIN
            ITEM GRAD'HS B = TRUE;
            ITEM BACH'DEG B = TRUE;
            ITEM MASTER'DEG B;
            END
      END
TABLE DETAIL'ID;
      BEGIN
      ITEM NAME C 20 = 'J SMITH';
      ITEM SOC'SEC'NO C 9;
      ITEM AGE U 7;
      TABLE EDUCATION = 2 (TRUE);
            BEGIN
            ITEM GRAD'HS B;
            ITEM BACH'DEG B;
            ITEM MASTR'DEG B;
            END
      END
```

Dimensioned tables nested within other dimensioned tables may be preset as well. Consider the following example:

```
TABLE LIBRARY(1 : 100);
      BEGIN
      ITEM AUTHOR C 20 = 20(' '),2('SMITH'), 'JONES';
      TABLE BOOKS(1 : 3);
            BEGIN
            ITEM TITLE C 10 = 60(' '), "VOL1', 'VOL2', "VOL3',
                              'VOL4', 'VOL5', 'VOL6', 'EARTH';
```

```
        ITEM DATE C 8;
        ITEM PAGES U = 60(0), 6(210), 432;
        END
    ITEM SHELF C 3 = 20(' '), 3('7H1');
    ITEM USED B;
    END
```

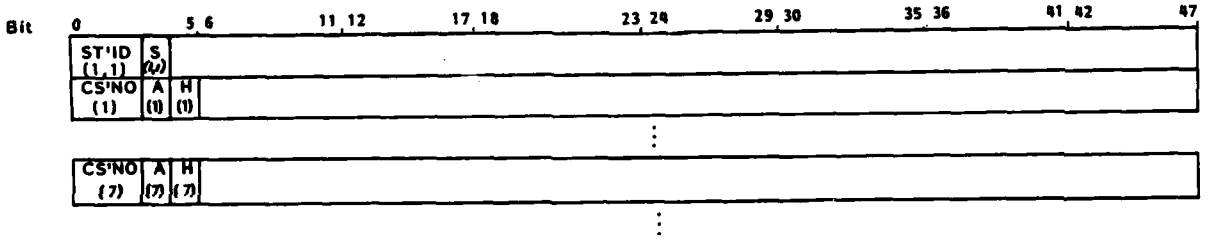
The table-preset given above skips the first twenty AUTHORS and presets the next two to SMITH and the following AUTHOR to JONES. By skipping sixty titles (twenty AUTHORs times three BOOKS each), AUTHOR SMITH has six books with TITLEs and PAGES preset and AUTHOR JONES has only one book with TITLE preset to EARTH and PAGES preset to 432. By skipping twenty of item SHELF, SMITH and JONES both have their books on SHELF 7H1.

NESTED TABLES: PACKING AND STRUCTURE

The packing and structure attributes discussed in Workbook 9 also apply to nested tables. Consider the following examples:

1.
```
TABLE SCHEDULE(1 : 4, 1 : 5) D;
    BEGIN
    ITEM STUD'ID U 4;
    ITEM SEX U 1;
    TABLE COURSES (1 : 7);
        BEGIN
        ITEM COURSENO U4;
        ITEM AUDIT B;
        ITEM HONORS B;
        END
    END
```

If BITSINWORD = 48, the above table may be laid out in the following way:



2. TABLE SCHEDULE(1 : 4, 1 : 5) D;

BEGIN

ITEM STUD'ID U 4;

ITEM SEX U 1;

TABLE COURSES (1 : 7) T;

BEGIN

ITEM COURSENO U 4;

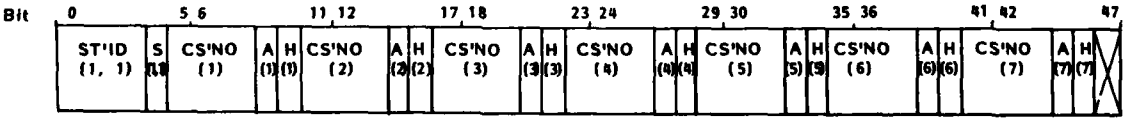
ITEM AUDIT B;

ITEM HONORS B;

END

END

If BITSINWORD = 48, the above table may be laid out in the following way:



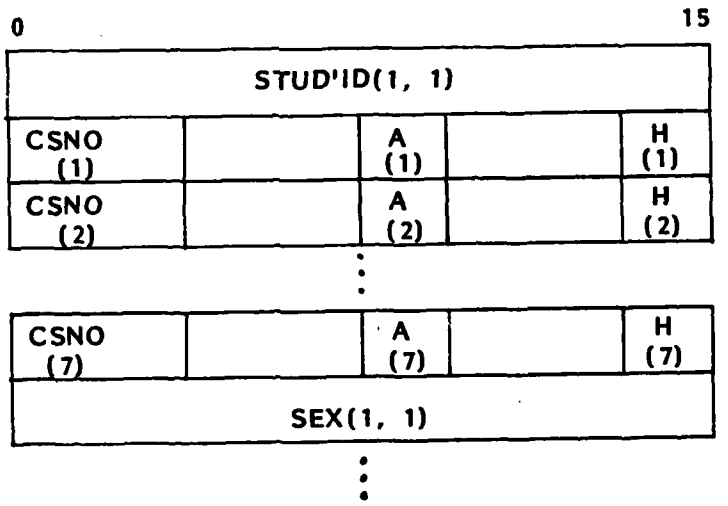
SPECIFIED NESTED TABLES

A nested table may be a specified table and appear within another specified table.

Example

```
TABLE SCHEDULE (1 : 4, 1 : 5) W 9;
        BEGIN
        ITEM STUD'ID U 4 POS(*, 0);
        ITEM SEX U 1 POS(*, 8);
        TABLE COURSES (1 : 7) W 1;
                BEGIN
                ITEM COURSENO U 4 POS(0, 0);
                ITEM AUDIT B POS(8, 0);
                ITEM HONORS B POS(15, 0);
                END
        END
```

If BITSINWORD = 16, the above table may be laid out in the following way:



CASE-VARIANT

A table sometimes involves a common part and a special part, whose form depends on information in the common part.

To get this form of a table, a case-variant is given as the last item in an entry-description.

Example

```
TABLE INFO;
        BEGIN
        ITEM NAME C 20;
        ITEM AGE U 7;
        ITEM GRAD'HS B;
        CASE GRAD'HS
                BEGIN
                (TRUE)   :    ITEM GRAD'U B;
                              ITEM MASTERS B;
                              ITEM PHD B;
                (FALSE)  :    ITEM LAST'GRADE U, 9 : 11;
                END
        END
```

A case-variant has the form:

```
        CASE  variant selector
                BEGIN
                [ default-variant ]
                variant
                ...
                END
```

The form of a variant is:

(case-index) declaration ...

At any given time, only one variant is present. Which variant is present depends on the value of variant-selector. Enough space is allocated to hold the largest variant.

> *NOTE: Reference to a component of a variant that is not present causes a run-time error.*

The value of the variant-selector indicates which variant is present. Variant-selector must be an item declared in an entry-description in the common part of the table that contains the case-variant. It can be of type integer, bit, character, or status.

The case-indices indicate the entry that is present for a particular value of the variant-selector. The types of the case-indices must be equivalent or implicitly convertible to the type of the variant-selector. A case-index may be a single value, a pair of bounds, or any combination of the two separated by commas. The form of a pair of bounds is:

first-case : last-case

A pair of bounds may be given only for type integer or status.

If the value of variant-selector does not match a case-index, the default-variant, if one is given, is selected. The form of the default-variant is:

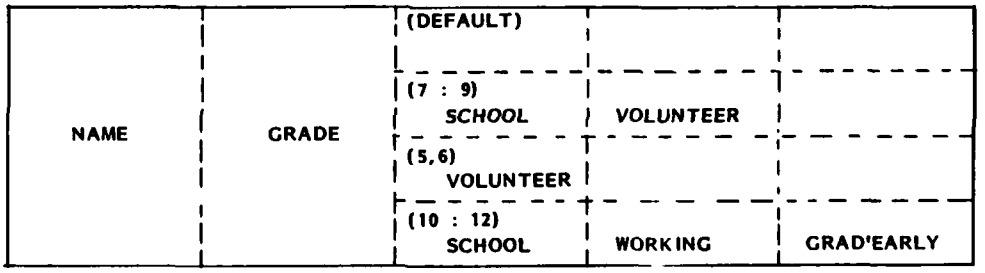
(DEFAULT) : declaration ...

Example

```
TABLE EDUC;
    BEGIN
    ITEM NAME C 20;
    ITEM GRADE U, 1 : 12;
```

```
CASE GRADE
        BEGIN
        (DEFAULT): ;
        (7 : 9):        ITEM SCHOOL C 5;
                        ITEM VOLUNTEER B;
        (5, 6):         ITEM VOLUNTEER B;
        (10 : 12):      ITEM SCHOOL C 5;
                        ITEM WORKING B;
                        ITEM GRAD'EARLY B;
        END
END
```

Table EDUC could be diagrammed as follows:



NOTE: The variant-options all occupy the same place in memory. If the value of the case-selector is changed, the programmer may make no assumptions as to the value in any of the case-options.

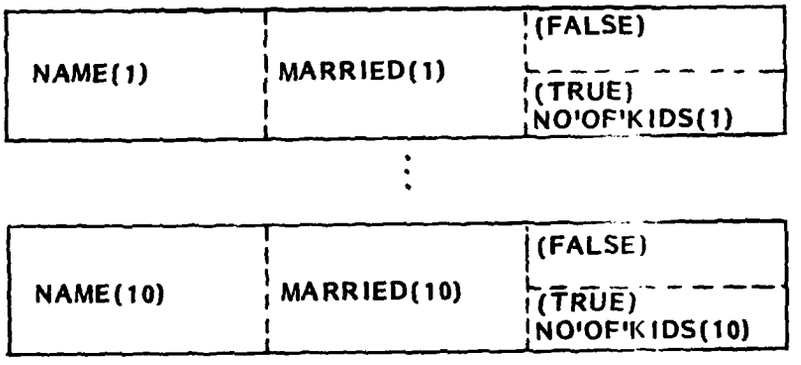
VARIANT ENTRIES IN DIMENSIONED TABLES

A dimensioned table may include a variant-option.

Example

```
TABLE INFO (1 : 10);
        BEGIN
        ITEM NAME C 20;
        ITEM MARRIED B;
        CASE MARRIED
                BEGIN
                (FALSE): ;
                (TRUE):  ITEM NO'OF'KIDS U, 0 : 20;
                END
        END
```

This table may be diagrammed as follows:



CASE-VARIANT PRESETS

A table-preset may be used to preset the items in a table with a variant-option. Using table EDUC, declared above, the preset could be given as a part of the table heading --

```
TABLE  EDUC  =  'J SMITH', 8, 'LMJHS',  TRUE;
```

The second preset value, GRADE = 8, makes the items SCHOOL and VOLUNTEER known and allows them to be preset. If the preset on GRADE had been missing, any attempt to preset any items in the case-variant would cause an error to be signalled.

Presets may also be given for variant entries which appear in dimensioned tables. For example:

```
TABLE INFO (1 : 10) ;
      BEGIN
      ITEM NAME C 20;
      ITEM MARRIED B = 5(TRUE, FALSE);
      CASE MARRIED
            BEGIN
            (FALSE):  ;
            (TRUE):  ITEM NO'OF'KIDS U, 0 : 20 =
                     (1, 3, 2, 6) ;
            END
      END
```

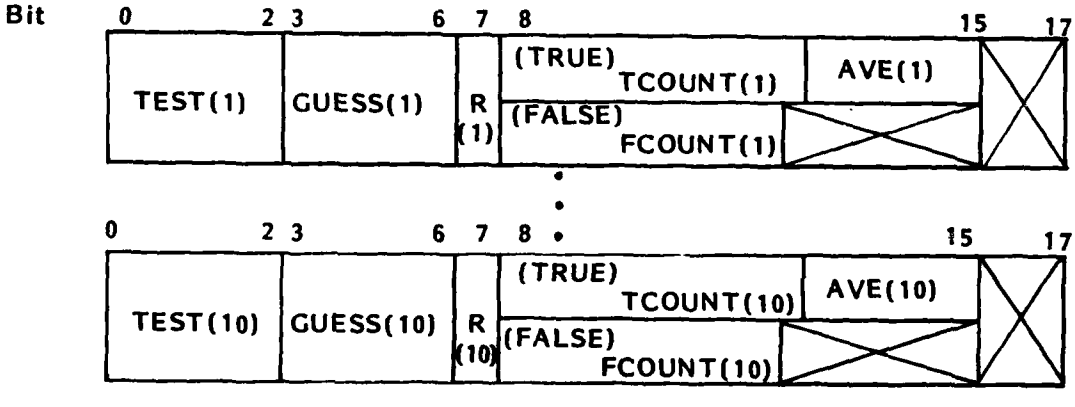
The table-preset given above presets the odd indexed items MARRIED to TRUE. The corresponding NO'OF'KIDS are preset to 1, 3, 2, 6. NO'OF'KIDS(9) is not preset.

VARIANT-OPTIONS: PACKING AND STRUCTURE

The packing and structure attributes discussed in Workbook 9 may also be used for variant-options. Consider the following examples:

1.
```
TABLE SAMPLE (1 : 10) D;
    BEGIN
    ITEM TEST U 3;
    ITEM GUESS U 4;
    ITEM RESULT B;
    CASE RESULT
        BEGIN
        (TRUE)   :   ITEM TCOUNT U 5;
                     ITEM AVE U 3;
        (FALSE)  :   ITEM FCOUNT U 4;
        END
    END
```

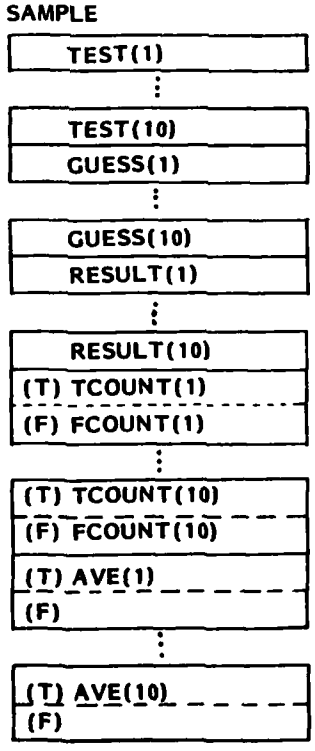
If BITSINWORD = 18, the above table may be laid out in the following manner:



2. TABLE SAMPLE(1 : 10) PARALLEL;

```
        BEGIN
        ITEM TEST U 3;
        ITEM GUESS U 4;
        ITEM RESULT B;
        CASE RESULT
            BEGIN
            (TRUE)  :   ITEM TCOUNT U 5;
                        ITEM AVE U 3;
            (FALSE) :   ITEM FCOUNT U 4;
            END
        END
```

Table SAMPLE could be diagrammed as follows:



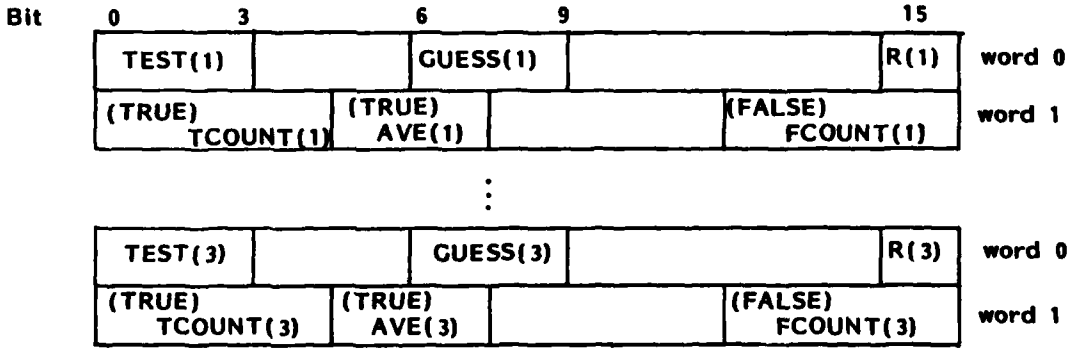
VARIANT-OPTIONS: SPECIFIED TABLES

Variant-options may also appear in specified tables.

Example

```
TABLE SAMPLE(1 : 3) W 2;
      BEGIN
      ITEM TEST U 3 POS(0, 0);
      ITEM GUESS U 4 POS(6, 0);
      ITEM RESULT B POS(15, 0);
      CASE RESULT
            BEGIN
            (TRUE)  :   ITEM TCOUNT U 5 POS(0, 1);
                        ITEM AVE U 3 POS(5, 1);
            (FALSE) :   ITEM FCOUNT U 4 POS(12, 1);
            END
      END
```

If BITSINWORD = 16, the above table may be laid out in the following way:



DATA REFERENCES

Nested Tables: A reference to an item in a nested table must be fully qualified; all tables enclosing that item must be given beginning with the outer-most table. Given the following table declaration:

```
TABLE CREDIT;
        BEGIN
        ITEM NEW'BOOK C 20;
        ITEM AUTHOR C 20;
        ITEM DATE C 8;
        TABLE FIRST'BOOK;
                BEGIN
                ITEM DATE C 8;
                TABLE OTHER'INFO;
                        BEGIN
                        ITEM PUBLISHER C 20;
                        ITEM ILLUS B;
                        END
                END
        END
```

The following data references are correct:

```
CREDIT.DATE
CREDIT.FIRST'BOOK.DATE
NEW'BOOK
CREDIT.FIRST'BOOK.OTHER'INFO.ILLUS
```

These fully qualified items may be used anywhere a simple item name may be used.

The form of a data reference to an item in a nested table is:

qualification name (subscript-list)

If a dimensioned table is nested within another dimensioned table, the outer most subscripts are given first, then the next ones in, etc.

For example, given the declaration of table LIBRARY,

```
TABLE LIBRARY(1 : 100);
      BEGIN
      ITEM AUTHOR C 20 = 20(' '),2('SMITH'), 'JONES';
      TABLE BOOKS(1 : 3);
            BEGIN
            ITEM TITLE C 10 = 60(' '), 'VOL1', 'VOL2', 'VOL3',
                              'VOL4', 'VOL5', 'VOL6', 'EARTH';
            ITEM DATE C 8;
            ITEM PAGES U = 60(0), 6(210), 432;
            END
      ITEM SHELF C 3 = 20(' '), 3('7H1');
      ITEM USED B;
      END
```

The following are correct data references:

```
AUTHOR (21)
LIBRARY.BOOKS.TITLE(21,2)
LIBRARY.BOOKS.PAGES(23,1)
SHELF(22)
```

These data references may be used anywhere a simple item name may be used.

As a further example, consider table DETAIL'ID:

```
TABLE DETAIL'ID (1 : 3, 1 : 5);
        BEGIN
        ITEM NAME C 20;
        ITEM SOC'SEC'NO C 9;
        ITEM AGE U 7;
        TABLE EDUCATION;
                BEGIN
                ITEM GRAD'HS B;
                ITEM BACH'DEG B;
                ITEM MASTR'DEG B;
                END
        END
```

The following are correct references:

```
NAME(2,3)
AGE(1,5)
DETAIL'ID.EDUCATION.GRAD'HS(2,3)
DETAIL'ID.EDUCATION.MASTR'DEG(1,5)
```

These data references may be used anywhere a simple item name may be used.

Example

```
IF DETAIL'ID.EDUCATION.GRAD'HS AND CREDITS = 140;
        DETAIL'ID.EDUCATION.BACH'DEG = TRUE;
```

Variant Entries: Given the following table declaration:

```
TABLE CHART;
        ITEM AVE'TEMP A 8, 3;
        ITEM AVE'RAIN A 8, 7;
        ITEM SEASON STATUS (V(WINT), V(SPRG), V(SUMR),
          V(FALL));
        CASE SEASON
                BEGIN
                (DEFAULT): ;
                (V(WINT)):  ITEM BELOWFRZ U;
                (V(SUMR)) : ITEM ABOVE100 U:
                END
```

Immediately following the statement,

```
SEASON = V(WINT);
```

the following items are made available: AVE'TEMP, AVE'RAIN, SEASON, and BELOWFRZ. If a reference is made to ABOVE100, an error is signalled.

If SEASON is set to V(SPRG), any reference to either BELOWFRZ or ABOVE100 will cause an error to be signalled.

An item in a variant entry may be used anywhere a simple item may be used.

Summary: An item in an undimensioned, uptyped table may simply be named.

An item in a dimensioned, untyped table must have a subscript-list.

An item in an undimensioned, typed table must be qualified, possibly using dot-qualification.

An item in a dimensioned, typed table must be qualified, possibly using dot-qualification, and must have a subscript-list.

An item in nested, typed or untyped, undimensioned table must be fully qualified.

An item in a nested, typed or untyped, dimensioned table must be fully qualified and subscripted.

Errors signalled:

- Reference to an item in a presently inactive variant
- Value of subscript out of range of dimension list.

TABLES IN RELATIONAL EXPRESSIONS

A relational operator compares two operands. The result of a relational expression is a Boolean value. The value 1B'1' represents the Boolean literal TRUE and the value 1B'0' the Boolean literal FALSE.

The relational operators are:

Operator	Meaning
=	Equals
<	Less than
>	Greater than
<>	Not equal
<=	Less than or equal to
>=	Greater than or equal to

The operands in a relational expression using any of the above operators must be both of the same type class. They may be integer-formulae, floating-formulae, fixed-formulae, character-formulae, status-formulae, pointer-formulae, or (in JOVIAL (J73/C)) **table-formula.**

Integer, floating, and fixed comparisons are made on the basis of the value of the operands. Character comparisons are made on the basis of the collating sequence of the character set for a given implementation. Status comparisons are made on the basis of the representation of the status values. Pointer comparisons are made on a target machine dependent basis.

Only the operators equals (=) and not equals (<>) may be used with bit operands.

Tables may also be compared for equality using the equals and not equals operators. Table comparisons are made on the basis of the bit representations of the tables. If the bit representations are identical, the tables are equal. Otherwise, they are not equal.

SECTION 4

SUBROUTINES

SUBROUTINES

A subroutine is an algorithm (either a procedure or a function) that may be invoked from more than one place in a program. They were discussed in Workbook 6. JOVIAL (J73/C) provides the following five extensions to JOVIAL (J73) relating to subroutines:

- the absolute address attribute
- the INTERRUPT attribute
- asterisk length character declarations
- specified parameter binding
- the READONLY declaration

These extensions are discussed below.

ABSOLUTE ADDRESS ATTRIBUTE

The absolute-address-attribute gives the machine address at which the subroutine is to be located.

The form is:

```
PROC  name  [ use-attribute ]  [ ( formal-list) ]
  [ item-type-description ]  [ absolute-address ] ;

      subroutine-body
```

Example

```
PROC  IFACT  (ARG)  U  POS(2200) ;

      subroutine-body
```

Everytime the IFACT function is invoked, the entry to the code is found at machine address 2200.

INTERRUPT USE-ATTRIBUTE

The INTERRUPT use-attribute (given directly following the procedure-name) may be given only in a non-nested procedure.

If INTERRUPT is specified, the interrupt-name is used to place the procedure so that it is accessible to the target machine interrupt hardware described by the character string.

The form is:

INTERRUPT interrupt-name

Interrupt-name is a character formula known at compile-time.

The INTERRUPT use-attribute should be used only for procedures that are part of the executive or operating system of a particular implementation.

A function may not be declared with the interrupt-attribute.

ASTERISK LENGTH CHARACTER DECLARATIONS

A character item that is a formal parameter may be delcared with asterisk size.

Example

```
PROC SOME'MESSAGE (DATE, DISTRIBUTION, TOPIC : MEMO);
    BEGIN
    ITEM DATE C 8;
    ITEM DISTRIBUTION U;
    ITEM TOPIC C 15;
    ITEM MEMO C *;
    .
    . executable statements
    .
    END
```

The procedure SOME'MESSAGE may be called with a date, a desired distribution list, and a topic. From that information, the procedure will generate a memo of any length as the procedure's output. The length of the memo will be that of the actual parameter corresponding to MEMO.

SPECIFIED PARAMETER BINDING

A formal parameter may specify the way in which it is bound to its corresponding actual parameter. The form is:

[parameter-binding] parameter-name [register-binding]

Parameter-binding is one of the following:

BYVAL - indicates values binding

BYREF - indicates reference binding

Register binding indicates that the parameter is to be stored in a register rather than in storage. The register-binding attribute has the form:

REGISTER target-register

Target-register is a character compile-time-formula that identifies the register. It is machine dependent.

Examples

Consider the following declaration:

```
PROC FIGURE (ADDTAB1, ADDTAB2 : SUMTAB, COUNT);
    BEGIN
    TABLE ADDTAB1 (1 : 10);
        ITEM ADD1 U;
    TABLE ADDTAB2 (1 : 10);
        ITEM ADD2 U;
    TABLE SUMTAB (1 : 10);
        ITEM SUM U;
```

```
ITEM COUNT U;
FOR I : 1 TO 10;
        FOR J : 1 TO 10;
                BEGIN
                SUM(I, J) = ADD1(I, J) + ADD2(I, J);
                COUNT = COUNT + 1;
                END
END
```

No parameter or register binding is specified; the default semantics hold, (the three tables are passed by reference, the output item - by value-result).

Using the procedure-definition above, consider the following:

```
PROC FIGURE(ADDTAB1, BYVAL ADDTAB2 : BYVAL SUMTAB2,
  BYREF COUNT);
```

The table ADDTAB1 will be passed by reference (default). The table ADDTAB2 will be passed by value (copied-in). The table SUMTAB2 will be passed by value-result (copied-in and copied-out). The item COUNT will be passed by reference.

```
PROC FIGURE(ADDTAB1, ADDTAB2 : BYVAL SUMTAB TBL1,
  BYREF COUNT GPR1);
```

The table SUMTAB will be passed by value-result. The value of the table will be stored in register TBL1. The item COUNT will be passed by reference. The address of the item will be stored in register GPR1.

NOTES: If register-binding is specified, parameter-binding must also be specified.

It is illegal to specify BYVAL for a parameter value too large to be contained in a register.

THE READONLY-DECLARATION

The readonly-declaration may be given only in a subroutine. It asserts that the specified data may be accessed in the subroutine only for reading.

The form of the readonly-declaration is:

READONLY data-name ,... ;

Any attempt to change the value of a data object named in a readonly-declaration in the subroutine that contains the readonly-declaration is an error.

The data-names given in a readonly-declaration must be known in the scope of that declaration. If a data-name is a table or a block, the effect of the readonly-declaration extends to all the components of the table or block.

SECTION 5

BUILT-IN FUNCTIONS

SOFTECH

BUILT-IN FUNCTIONS

JOVIAL (J73/C) provides four additional built-in functions:

- NENT
- FIRST
- LAST
- NEW

NENT

The NENT function returns the number of entries in the table or table-type given as its argument.

The form is:

NENT (argument)

The type returned is S with default size.

Example

Given the following declarations:

```
TYPE DIMENSIONS TABLE
        BEGIN
        ITEM LENGTH U;
        ITEM HEIGHT U;
        ITEM WIDTH U;
        END
TABLE COMPONENTS(10, 5) DIMENSIONS;
NENT(DIMENSIONS)    returns 1
NENT(COMPONENTS)    returns 66
```

INVERSE FUNCTIONS

Status Types:

Just as in JOVIAL (J73), the inverse functions find the lowest and highest values of the status-list associated with a status argument.

The inverse functions are:

FIRST ({ status-variable / status-type })

The inverse function that find the highest value has the form:

LAST ({ status-variable / status-type })

The type of the result is the same as the type of the argument.

Example

```
TYPE LETTER (2V(A), 3V(B), 1V(C), 8V(D), 4V(E));
      ANSWER = FIRST(LETTER);
      RESULT = LAST(LETTER);
      ANSWER is assigned V(C) and RESULT is assigned V(D).
```

Numeric Types:

In J73/C, the inverse functions find the lowest and highest values of the item associated with the numeric argument as well:

The inverse function that finds the lowest value has the form:

FIRST ({ numeric-variable / numeric-type-name })

The inverse function that finds the lowest value has the form:

LAST ({ numeric-variable / numeric-type-name })

The type of the result is the same as the type of the argument.

NOTES: Signed and unsigned integer, floating point, and fixed point are all permissible numeric types.

Example

```
ITEM VALUE F 15, -6.75 : 49.25;

FOR I : FIRST(VALUE) BY .25 TO LAST (VALUE);
```

NEW FUNCTION

The NEW function is used in connection with dynamic storage allocation to obtain storage.

The form of the NEW function is:

NEW (table-type-name)

The table-type-name given as an argument may or may not be associated with a zone. If it is not, a call on the NEW function allocates storage for a data object of the specified type in heap storage and returns a pointer to that newly allocated object.

If the table-type-name is associated with a zone, a call on the NEW function invokes the ALLOC function defined for the associated zone, implicitly passing the WORDSIZE of the table-type-name argument in addition to the table-type-name. The ALLOC function returns an untyped pointer, which is implicitly converted to a typed pointer and is the NEW function return value.

The type of the value returned is a pointer with a type-name attribute of the table-type-name argument.

SECTION 6

DIRECTIVES

DIRECTIVES

Directives are used to provide supplemental information to the compiler about the program. Directives affect output format, program optimization, data and subroutine linkage, debugging information, and other aspects of program processing. JOVIAL (J73/C) provides two additional directives - the compoolrefsonly-directive and the characters-directive.

COMPOOLREFSONLY-DIRECTIVE

The compoolrefsonly-directive is used to restrict the use of REF-specifications to compools. If a procedure-module or a main-program-module has a compoolrefsonly-directive, the use of REF-specifications in that module is prohibited and the compiler will issue an error message if a REF-specification is encountered.

The form of this directive is:

```
!COMPOOLREFSONLY ;
```

The effect of using a compoolrefsonly-directive is to require that all inter-module communication occur via compools by the use of compool-directives.

The compoolrefsonly-directive must be given only after all compool-directives and before the text of the module. Only one compoolrefsonly-directive may be given in a module. A compoolrefsonly-directive in a compool-module has no effect.

CHARACTERS-DIRECTIVE

The characters-directive is supplied for the case in which an implementation has more than one supported character set. The characters-directive is used to specify which character set is to be used to represent character data. The form of the characters-directive is:

!CHARACTERS set-name ;

Set-name is a name defined by the implementation for each supported character set. An implementation may have several character sets or only one.

If a characters-directive is not given, an implementation-dependent default character set is assumed.

The representations of the status-constants in the status-list of the built-in status type ALPHA are the bit patterns associated with the characters in the character set indicated by either the explicit or default characters-directive. For each supported character set, an implementation must supply an ALPHA status-list.

The size for ALPHA is considered to be the minimum number of bits needed to represent the values in the designated character set as unsigned integers.

The characters-directive may appear only following the word START at the beginning of a module, preceding another directive or text.

Only one characters-directive may be given in a module. All modules in a complete program must either have no characters-directive or must have the same characters-directive.

THE JOVIAL (J73) WORKBOOK

VOLUME 14

ZONES AND ENCAPSULATIONS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

Workbook 14 is intended for use with Tape 14 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

Section 1 discusses the declaration and use of zones in JOVIAL (J73/C). In addition the ALLOC function and the DEALLOC procedure are also addressed. An additional CHOL extension -- encapsulated data -- is treated in Section 2. Section 3 is a review of the material presented in this segment.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction- a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

ZONES

SOFTECH

ZONES

A zone is a special portion of storage that may be allocated, formatted, and deallocated by the programmer. Any number of zones may be declared. Each zone has a certain amount of storage and special subroutines that handle the allocation, formatting, and deallocation of that storage. The data that is allocated in a zone is data declared using a table-type-name that has a zone-part specified in its declaration.

ZONE-DECLARATION

A zone is declared as follows:

```
ZONE  zone-name [ zone-heading ] ;
      BEGIN
      zone-storage
      subroutine-definition ...
      END
```

The square brackets indicate that the zone-heading is optional. The sequence '...' indicates that one or more subroutine-definitions may be given in the zone-declaration.

A zone-declaration must have an ALLOC function and a DEALLOC procedure defined. The ALLOC function is implicitly invoked when the program calls the NEW function to allocate storage for a table of a type declared with a zone-part. The DEALLOC procedure is implicitly invoked when the program calls the FREE procedure to return storage for such a table.

Zone Heading

The zone-heading may include information about the allocation, the location in memory, and the number of words in the zone. The form of a zone-heading is:

[allocation-spec] [absolute-address] [W zone-size]

The square brackets indicate that all the components in the zone-heading are optional.

If an absolute-address is given, the zone is allocated starting at the specified address in memory. An absolute-address may be given only if the allocation of the zone is STATIC either explicitly or by default.

The zone-size in a zone-heading gives the number of words to be allocated for a zone. Zone-size is an integer formula known at compile-time. If zone-size is given, all tables in zone-storage must be specified tables.

Zones are represented as tables. Zones may contain specified table-options or ordinary table-options. The size of the zone is determined by the zone-size if the zone consists of specified table-options. Otherwise, the size of the zone is determined by the way the compiler chooses to allocate the ordinary table-options.

Given the following zone-declaration:

```
ZONE BIGZONE;
        BEGIN
        TABLE SPACE (1 : 1000);
        ITEM ONESPACE U;
        PROC ALLOC (   ) P;
              ...
        PROC DEALLOC (  ,  ):
              ...
        END
```

Storage for zone BIGZONE requires 1000 words.

Zone Storage

Zone-storage is expressed as one or more table-options. The declared tables may be used to simply save space that may be allocated under a format to be specified later. The form is:

table-option ...

Given the following zone-declaration:

```
ZONE SMALLZONE;
        BEGIN
        TABLE WORDS (1 : 100);
                ITEM DATA U;
        TABLE FLAGS (1 : 100) T;
                ITEM AVAIL B;
        PROC ALLOC (  ) P;
                ...
        PROC DEALLOC (  ,  );
                ...
        END
```

Storage for zone SMALLZONE requires of 100 words and 100 bits.

ZONE-PART IN A TABLE-TYPE-DECLARATION

A type that is to be associated with a zone has a zone-part in its declaration, as follows:

```
TYPE  table-type-name  [ zone-part ]
        TABLE  [ table-heading ];
                entry-description
```

Zone-part has the form:

IN zone-name

Table-heading consists of the dimension-list, structure-spec, like-option, packing-spec, or table-type-name; any of which is optional.

When the NEW function is used with a table-type-name declared with a zone-part, the compiler invokes the ALLOC function for that zone to allocate storage from the zone instead of from heap storage.

Given the following declarations:

```
ZONE MESSAGEPOOL;
        BEGIN
        TABLE SPACE (1 : 10000);
        ITEM ONESPACE U;
        PROC ALLOC (  ) P;
          ...
        PROC DEALLOC (  ,  );
          ...
        END
TYPE MESSAGE IN MESSAGEPOOL TABLE;
        BEGIN
        ITEM TEXT C 132;
        ITEM CODE U;
        END
TYPE STATISTICS TABLE;
        BEGIN
        ITEM COUNT U;
        ITEM FREQUENCY F;
        END
ITEM MESSAGEPTR P MESSAGE;
ITEM STATPTR P STATISTICS;
```

When the NEW function is called with the table-type-name STATISTICS, the storage for that type is supplied from heap storage. When the NEW function is called with the table-type-name MESSAGE, the ALLOC function associated with zone MESSAGEPOOL is invoked.

Similarly, when the FREE procedure is called with the actual parameter STATPTR, storage is returned to heap storage. When the FREE procedure is called with the actual parameter MESSAGEPTR, the DEALLOC procedure associated with zone MESSAGEPOOL is invoked and the actions dictated by that subroutine are performed.

ALLOCATION

A zone-declaration must have an ALLOC function and a DEALLOC procedure.

When the NEW function is called with a table-type-name that has a zone-part, the ALLOC function in the designated zone-declaration is invoked to handle the allocation. When the FREE procedure is called with an actual parameter that points to an object whose type is associated with a zone, the DEALLOC procedure in that zone is invoked to handle the deallocation.

The ALLOC Function

The ALLOC function must have exactly one parameter, an input parameter of type integer. The size and range of this parameter must be large enough to hold the WORDSIZE of the largest object to be allocated in that zone.

When the NEW function is called with a table-type-name that has a zone-part, the WORDSIZE of a table of the type of the actual parameter of NEW is implicitly passed to the formal parameter. The value returned by ALLOC is the value of the NEW function. The type of that value is implicitly converted to P table-type-name, where table-type-name is the actual parameter of NEW.

For example, the NEW function may be called as follows:

```
NEW (MESSAGE)
```

MESSAGE is a table-type-name with a zone-part. The programmer-supplied ALLOC function in MESSAGEPOOL is invoked, and its return value, a pointer to an object of type MESSAGE, becomes the value of NEW.

An example of an ALLOC function is given later in this section.

The DEALLOC Procedure

The DEALLOC procedure must have exactly two input parameters. The first parameter must be an untyped pointer and the second must be an integer whose size and range must be large enough to hold the WORDSIZE of the largest object to be allocated in that zone.

When the FREE procedure is called with an actual parameter that points to an object whose type is associated with a zone, the address of the actual parameter is implicitly passed to the first formal parameter of the DEALLOC procedure, and the WORDSIZE of the type associated with the pointer argument of FREE is implicitly passed to the second formal parameter.

For example, the FREE procedure may be called, as follows:

```
FREE (MESSAGEPTR);
```

MESSAGEPTR is a pointer that points to an object of type MESSAGE. The type MESSAGE is declared with a zone-part. The programmer-supplied DEALLOC procedure in MESSAGEPOOL is invoked. The address of MESSAGEPTR is passed as the first parameter and the WORDSIZE of MESSAGE, the type associated with MESSAGEPTR, is passed as the second parameter.

An example of a DEALLOC procedure is given later in this section.

ZONE INITIALIZATION

If the declaration of the zone includes an INITIALIZE subroutine, that subroutine is invoked implicitly after storage for the zone is allocated and before execution of any of the statements in the scope containing the zone declaration.

An example of an INITIALIZE subroutine is given below.

Zones: An Example

The following is a simple example of some storage management subroutines declared for zone DATA.

```
ZONE DATA
      BEGIN
      TABLE SPACE (1 : 1000);
            ITEM WORD U;
      TABLE FLAG (1 : 1000) T;
            ITEM AVAIL B;

      PROC ALLOC (XX) P;
            BEGIN
            ITEM XX U;
            FOR I : 1 TO 1000;
                  IF AVAIL (I);
                        BEGIN
                        FOR J : I TO I + XX - 1;
                              IF NOT AVAIL (J);
                                    GOTO NEXTWORD;
                        ALLOC = LOC (WORD(I)));
                        FOR J : I TO I + XX - 1;
                              AVAIL (J) = FALSE;
                        RETURN;
                        NEXTWORD :
                        END
            ERROR (FULL);
            END

      PROC DEALLOC (PTR, XX);
            BEGIN
            ITEM PTR P;
            ITEM XX U;
            ITEM INDEX U;
            INDEX = (U (PTR) - U (LOC (SPACE))) / LOCSINWORD;
            FOR I : INDEX TO INDEX + XX - 1;
                  AVAIL (I) = TRUE;
            END

      PROC INITIALIZE;
            FOR I : 1 TO 1000;
                  AVAIL (I) = TRUE;
      END
```

The zone DATA contains two tables, each of which has 1000 entries. Table SPACE contains 1000 unsigned integers; table FLAG contains Boolean flags that indicate whether the corresponding entry in table SPACE is available or unavailable.

The INITIALIZE procedure is implicitly invoked after storage for zone DATA is allocated. It sets all the entries in table FLAG to TRUE, to indicate that the corresponding word in table SPACE is available.

The ALLOC function is implicitly invoked when space in the zone is requested. It searches table FLAG until it finds an available entry, and searches starting at the entry and continuing for the number of words needed to see if enough space is available to satisfy the allocation request. If so, it returns a pointer to the first word of the allocated space, and flags the corresponding entries in table FLAG FALSE to indicate that the space is no longer available. If enough space is not found, ALLOC continues searching. If no available storage is found, an error subroutine is called.

The DEALLOC procedure is implicitly invoked when space is to be returned to the zone. It locates the appropriate index into table FLAG and sets the number of flags indicated to TRUE to show that the corresponding words in table SPACE are available.

These routines do not physically allocate or deallocate storage. Instead, they manipulate storage within the zone, which was allocated by the zone-declaration.

Consider a use of storage allocated from the above zone:

```
TYPE READING IN DATA TABLE;
        BEGIN
        ITEM LAT A 8,7;
        ITEM LONG A 8,7;
        ITEM TIME C 4;
        ITEM DATE U;
        END
TYPE SPECS IN DATA TABLE;
        BEGIN
        ITEM VOLUME C 4;
        ITEM CODE B BITSINWORD;
        END
ITEM READPTR P READING;
ITEM SPECPTR P SPECS;
```

The use of the NEW function with the table-type-name READING, which is declared with a zone-part, invokes the ALLOC function declared for the appropriate zone, in this case, zone DATA.

```
READPTR = NEW (READING);
```

The WORDSIZE of the table-type READING, 4 if BYTESINWORD is 4, is passed to the ALLOC function. The ALLOC function searches table FLAG and returns a pointer to a block of four unsigned integers in the zone DATA if space is available. The pointer returned by the ALLOC function is implicitly converted to a pointer of type READING and is the value of the NEW function.

The use of the NEW function with the table-type-name SPECS, which is declared with a zone-part, invokes the ALLOC function declared for the appropriate zone, in this case, zone DATA.

```
SPECPTR = NEW (SPECS);
```

The WORDSIZE of the table-type SPECS, 2 if BYTESINWORD is 4, is passed to the ALLOC function. The ALLOC function searches table FLAG and returns a pointer to a block of two unsigned integers in zone DATA if space is available. The pointer returned by the ALLOC function is implicitly converted to a pointer of type SPECS and is the value of the NEW function.

Similarly, the use of FREE with a pointer associated with a type declared with a zone-part returns storage to the appropriate zone. For example, the function-call FREE(READPTR) releases four words starting at the indicated position in zone DATA, and the function-call FREE(SPECPTR) releases two words, starting at the indicated position in zone DATA.

The implicit pointer conversions make it unnecessary for the types of the tables allocated by NEW (and ALLOC) or their fields to be equivalent to the tables declared in the type of the zone-storage.

SECTION 2

ENCAPSULATED DATA

SOFTECH

ENCAPSULATED DATA

JOVIAL (J73/C) provides a capability that allows a data object to be described in terms of the operations that may be performed on that data rather in terms of its physical structure. An encapsulated data object is a special kind of table whose components are not available for access outside the encapsulation unless they are explicitly designated as being available. The encapsulation may also contain definitions of subroutines to manipulate the data. This process of identifying data within an encapsulation as available for use outside the encapsulation is called exportation.

An encapsulation may export data-names and subroutines-names. Exported data may be designated as being available for reading only, for writing only, or for both. Exported subroutines may manipulate the data within the encapsulation. If only subroutines are exported, the data within the encapsulation may be accessed only through the use of these subroutines.

ENCAPSULATED DECLARATIONS

An encapsulation may be given in either a table- or a table type-declaration. In a table-declaration the encapsulation follows the table-name, as follows:

TABLE table-name encapsulation

In a table type-declaration, the encapsulation follows the table-type-name, as follows:

TYPE table-type-name encapsulation

As with other data, declaring an encapsulation in a table-declaration produces a data object, while declaring an encapsulation in a type-declaration declares a template that may be used in table-declarations.

The form of an encapsulation is:

```
ENCAPSULATION ;
BEGIN
EXPORTS  exported-component ,... ;
[ type-declaration ... ]
entry-description
subroutine-definition ...
END
```

One or more exported-components, separated by commas, may be given. Any number of subroutine-definitions may be given.

Names of types, items, tables, and subroutines declared or defined within an encapsulation are not available outside the encapsulation unless they are exported.

In the simplest case, an exported-component is simply a name. A name given as an exported-component must be declared within the encapsulation.

For example:

```
TABLE STACK ENCAPSULATION;
      BEGIN
      EXPORTS POP, PUSH, INIT;
      TABLE TSTACK (1 : 1000);
            ITEM VALUE U;
      ITEM TOP U;
      PROC INIT;
            TOP = 0;
      PROC PUSH (VAL);
            BEGIN
            ITEM VAL U;
            TOP = TOP + 1;
            VALUE (TOP) = VAL;
            END
```

```
PROC POP (: VAL);
        BEGIN
        ITEM VAL U;
        VAL = VALUE (TOP);
        TOP = TOP - 1;
        END
END
```

This encapsulation implements a stack. It exports the names of three subroutines, INIT, POP, and PUSH. The actual physical representation of the stack is irrelevant to the programmer using the exported subroutines. As long as the interface (the calling sequence) to the subroutines does not change, the physical representation of the stack may be altered without affecting the programmer's use of the encapsulated data.

Kinds of Access

The kind of access that may be made to the exported-component outside the encapsulation may be given before the exported-component, as follows:

[access] name

Two forms of access may be specified:

READONLY

WRITEONLY

If READONLY access is specified, the name may only be read; it may not be used in any context in which its value is changed; for example, as the target of an assignment-statement or as an output parameter. If WRITEONLY access is specified, the name may only be written; it may not be used in any context in which its value is referenced. If access is not specified, the name may be both read and written.

The following is an example of export-access:

```
TABLE TAX ENCAPSULATION;
      BEGIN
      EXPORTS READONLY TAXRATE, CHANGERATE;
      TABLE TAXRATE (9);
            ITEM RATE F;
      PROC CHANGERATE (FACTOR);
            BEGIN
            ITEM FACTOR F;
            FOR I : 0 TO 9;
                  RATE (I) = (1 + FACTOR) * RATE (I);
            END
      END
```

The table TAXRATE is exported for READONLY access from the TAX encapsulation. It may be examined outside the encapsulation, but any change to TAXRATE must be made within the encapsulation by using the encapsulated subroutine CHANGERATE. For example:

```
IF RATE (0) < STANDARD;

      CHANGERATE (STANDARD - RATE (0));
```

Exported Nested Table Names

When a table-name is given as an exported-component, the name of the table and the names in the top level of the table's structure are exported. The name of an item or table within a nested table may not be exported unless the name of the enclosing table is also exported.

The with-phrase is used to export the names of entries in nested tables. It is given following the name of the enclosing table in an exported-component, as follows:

[access] name [WITH (exported-component ,...)]

The names given as exported-components in a with-phrase must be components of the nested table whose name precedes the WITH.

Given the following table encapsulation:

```
TABLE LEAGUE ENCAPSULATION;
        BEGIN
        EXPORTS TEAM;
        ITEM LEAGUENAME C 10;
        TABLE RULES (100);
                ITEM RULENO C 20;
        TABLE TEAM (10);
                BEGIN
                ITEM NAME C 15;
                ITEM CITY C 15;
                TABLE RECORD;
                        BEGIN
                        TABLE WINS;
                        BEGIN
                        ITEM SHUTOUTS U;
                        ITEM OTHER U;
                        END
                ITEM LOSSES U;
                END
        ITEM OWNER C 15;
        END
...
END
```

The exported-component is the name TEAM, the name of a table, so the names of the top level structure of that table (NAME, CITY, RECORD, and OWNER) are also exported.

To make available the names of the top level within the nested table RECORD, the table RECORD must be exported with a with-phrase, as follows:

```
EXPORTS TEAM, RECORD WITH (WINS, LOSSES);
```

To make available the name SHUTOUT, the following may be written:

```
EXPORTS TEAM, RECORD WITH (WINS WITH (SHUTOUTS), LOSSES);
```

ENCAPSULATED SUBROUTINES

Encapsulated data may be accessed by subroutines declared within the encapsulation. Such subroutines are called encapsulated subroutines.

If an encapsulated subroutine-name is exported, the subroutine may be used to manipulate the data that is not exported.

Subroutines in a Table Encapsulation

The following is an example of a subroutine-definition in a table encapsulation:

```
TYPE MONEY A 12,2;
TABLE ACCOUNT ENCAPSULATION;
        BEGIN
        EXPORTS TRANSACT;
        ITEM SSNO C 12;
        ITEM BALANCE MONEY;
        PROC TRANSACT (INCR);
                BEGIN
                ITEM INCR MONEY;
                IF BALANCE + INCR < 0.0;
                        REPORT ('OVERDRAWN');
                ELSE
                        BALANCE = BALANCE + INCR;
                END
        END
```

The table encapsulation ACCOUNT exports the subroutine TRANSACT, which may be used to adjust BALANCE. The value of BALANCE is not available outside the encapsulation.

An example of the use of TRANSACT is:

```
        CASE DAY;
                BEGIN
                (DEFAULT): ;
                (1):   TRANSACT (-MORTGAGE);
                (15):  TRANSACT (SALARY);
                (28):  TRANSACT (-SAVINGS);
                END
```

This case-statement subtracts the mortgage from balance on the first day of the month, adds in the salary on the fifteenth day, and subtracts off the savings on the twenty-eighth day.

Subroutines in a Type Encapsulation

A subroutine in a type encapsulation must be written to handle the encapsulated data in a general way, because the type encapsulation may be used to declare many different tables. An actual table of the encapsulated type must be passed as one of the parameters to the subroutines associated with that encapsulated type.

Given the following type encapsulation:

```
TYPE ACCOUNT ENCAPSULATION;
        BEGIN
        EXPORTS TRANSACT;
        ITEM SSNO C 12;
        ITEM BALANCE MONEY:
        PROC TRANSACT (ACNT, INCR);
                BEGIN
                TABLE ACNT ACCOUNT;
                ITEM INCR MONEY;
                IF ACNT.BALANCE + INCR < 0.0;
                        REPORT ('OVERDRAWN');
                ELSE
                        ACNT.BALANCE = ACNT.BALANCE +
                        INCR;
                END
        END
```

The encapsulated type ACCOUNT may be used to declare any number of tables and the TRANSACT subroutine may be used by any of these tables to adjust the balance. The TRANSACT subroutine has been generalized by the inclusion of an additional formal parameter ACNT, which is declared to be of the encapsulated type ACCOUNT. The subroutine is written using name qualification as shown above.

The following example uses the encapsulated type ACCOUNT:

```
TABLE SMITH ACCOUNT;
TABLE JONES ACCOUNT;

IF GOLD > 1000;
        BEGIN
        TRANSACT (JONES, -1000);
        TRANSACT (SMITH, 1000);
        END
```

The tables SMITH and JONES are both declared to have type ACCOUNT. The TRANSACT subroutine is used to subtract 1000 from the JONES account and add 1000 to the SMITH account if GOLD exceeds 1000.

If the encapsulated type is used to declare a dimensioned table, as follows:

```
TABLE BANK (100) ACCOUNT;
```

The TRANSACT subroutine may be used to add 10 to the balance of each account in BANK as follows:

```
FOR I : 0 TO 100;
        TRANSACT (BANK (I), 10);
```

INITIALIZING ENCAPSULATED DATA

Data objects in an encapsulation may not be initialized by a preset. These data objects must be initialized by a subroutine.

The encapsulation for a stack given earlier in this chapter illustrates initialization:

```
TABLE STACK ENCAPSULATION;
        BEGIN
        EXPORTS POP, PUSH, INIT;
        TABLE TSTACK (1000);
                ITEM VALUE U;
        ITEM TOP U;
        PROC INIT;
                TOP = 0;
        PROC PUSH (VAL);
                BEGIN
                ITEM VAL U;
                TOP = TOP + 1;
                VALUE (TOP) = VAL;
                END
        PROC POP (: VAL);
                BEGIN
                ITEM VAL U;
                VAL = VALUE (TOP);
                TOP = TOP - 1;
                END
        END
```

The INIT subroutine sets the value of TOP to zero.

ASSIGNING VALUES TO ENCAPSULATED DATA

An encapsulated table may be given as the target of an assignment-statement only if the subroutine-name ASSIGN is exported. If the ASSIGN subroutine is defined within the encapsulation, it is invoked whenever the assignment operator is applied to an encapsulated table outside of the encapsulation. If the ASSIGN subroutine is not defined within the encapsulation, the default semantics (bit by bit copy) of table assignment apply.

By defining the ASSIGN subroutine within an encapsulation, the default semantics of the assignment-statement may be overridden. The ASSIGN subroutine is a procedure and it must have exactly one input parameter and one output parameter. These parameters must be of the encapsulated type.

The following example implements a stack using a linked list representation:

```
TYPE LSTACK ENCAPSULATION;
        BEGIN
        EXPORTS PUSH, POP, INIT, ASSIGN;
        TYPE USTACK TABLE;
                BEGIN
                ITEM VALUE U;
                ITEM LINK P USTACK;
                END
        ITEM HEADPTR P USTACK;
        PROC PUSH (DATA, STACK);
                BEGIN
                ITEM DATA U;
                TABLE STACK LSTACK;
                ITEM P1 P USTACK;
                P1 = NEW (USTACK);
                VALUE @ P1 = DATA;
                LINK @ P1 = STACK.HEADPTR;
                STACK.HEADPTR = P1;
                END
```

```
PROC POP (STACK) U;
        BEGIN
        TABLE STACK LSTACK;
        ITEM P1 P USTACK;
        P1= HEADPTR;
        POP = VALUE @ P1;
        STACK.HEADPTR = LINK @ P1;
        FREE (P1);
        END
PROC INIT (STACK);
        BEGIN
        TABLE STACK LSTACK;
        STACK.HEADPTR = NULL;
        END
PROC ASSIGN (STACKSOURCE : STACKTARGET);
        BEGIN
        TABLE STACKSOURCE LSTACK;
        TABLE STACKTARGET LSTACK;
        ITEM P1 P USTACK;
        ITEM P2 P USTACK;
        P2 = STACKTARGET.HEADPTR;
        FOR P1 : STACKSOURCE.HEADPTR THEN LINK
          @ P1 WHILE P1 <> NULL;
              BEGIN
              IF P2 = NULL;
                    BEGIN
                    P2 = NEW (USTACK);
                    LINK @ P2 = NULL;
                    END
              VALUE @ P2 = VALUE @ P1;
              P2 = LINK @ P2;
              END
        FOR P2 : LINK @ P2 THEN LINK @ P2 WHILE
          P2 <> NULL;
              FREE (P2);
        END
END
```

The INIT subroutine sets the HEADPTR of the designated stack to NULL to indicate an empty stack. The PUSH subroutine pushes an entry onto the stack. The POP subroutine pops an entry off the stack. The ASSIGN subroutine is used to set one stack equal to another. Given the following program fragment:

```
TABLE NEWDATA LSTACK;
TABLE OLDDATA LSTACK;
INIT (NEWDATA);
INIT (OLDDATA);
FOR I : 0 TO 100;
      PUSH (NEWDATA, SAMPLE (CODE));
OLDDATA = NEWDATA;
```

This fragment collects 101 sample values in the stack NEWDATA and then copies that data into the stack OLDDATA.

Normal Assignment for an Encapsulated Table

If the assignment of an encapsulated data object is to have the same semantics as the assignment operator, the name ASSIGN is given in the export-list but is not defined in the encapsulation.

Given the following encapsulation that implements a stack as a table:

```
TYPE TSTACK ENCAPSULATION;
      BEGIN
      EXPORTS POP, PUSH, INIT, ASSIGN;
      TABLE USTACK (1000);
            ITEM VALUE U;
      ITEM TOP U;
      PROC INIT (STACK);
            BEGIN
            TABLE STACK TSTACK;
            STACK.TOP = 0;
            END
      PROC PUSH (STACK, VAL);
            BEGIN
            TABLE STACK TSTACK;
            ITEM VALU U;
            STACK.TOP = STACK.TOP + 1;
            STACK.USTACK.VALUE (STACK.TOP) = VAL;
            END
      PROC POP (STACK : VAL);
            BEGIN
            TABLE STACK TSTACK;
            ITEM VAL U;
            VAL = STACK.USTACK.VALUE (STACK.TOP);
            STACK.TOP = STACK.TOP - 1;
            END
      END
```

Since the ASSIGN subroutine-name is given in the export-list, data objects of the encapsulated type may be used on the left hand side of an assignment-statement, as follows:

```
TABLE REQUESTS TSTACK;
TABLE BACKLOG TSTACK;

BACKLOG = REQUESTS;
```

Since the underlying representation of TSTACK is a table, the default assignment semantics for tables is appropriate, and the bit representation of the source table will be copied into the target table.

If the ASSIGN subroutine-name is not given in the export-list, an encapsulation may not be used as the target of an assignment-statement.

Assignment within the Encapsulation

Within the encapsulation, the assignment operator may be used with its usual meaning. The ASSIGN subroutine may also be used, but it must be called as a subroutine.

COMPARING ENCAPSULATED TABLES

An encapsulated table may be used in a relational expression only if the subroutine-name COMPARE is exported. If the COMPARE subroutine is defined within the encapsulation, it is invoked whenever a relational operator is applied to an encapsulated table outside of the encapsulation. If the COMPARE subroutine is not defined within the encapsulation, the default semantics (bit by bit comparison) of table comparison apply.

By defining the COMPARE subroutine within an encapsulation, the default semantics of the equals and not equals operators may be overridden. The COMPARE subroutine is a function and it must have exactly two input parameters. The type of these parameters must be of the encapsulated type. The COMPARE function returns a Boolean value, TRUE or FALSE.

The COMPARE function is invoked when tables of the encapsulated type are used in a relational expression outside of the encapsulation. A relational expression with table operands may use only the operators equals (=) and not equals (<>) operators. The left operand of the relational expression is passed to the COMPARE function as its first parameter, and the right operand is passed as the second parameter. If the relational expression contains the equals operator, the return value of the COMPARE function is taken as the value of the relational expression. If the relational expression contains the not equals operator, the return value of COMPARE is the negation.

Given the following COMPARE function for the encapsulation LSTACK given earlier:

```
PROC COMPARE (STACKL, STACKR) B 1;
        BEGIN
        TABLE STACKL LSTACK;
        TABLE STACKR LSTACK;
        ITEM P1 P USTACK;
        ITEM P2 P USTACK;
        P2 = STACKR.HEADPTR;
        FOR P1 : STACKL.HEADPTR THEN LINK @ P1 WHILE
          P1 <> NULL;
                BEGIN
                IF P2 = NULL;
                        BEGIN
                        COMPARE = FALSE;
                        RETURN;
                        END
                IF VALUE @ P1 = VALUE @ P2;
                        P2 = LINK @ P2;
                ELSE
                        BEGIN
                        COMPARE = FALSE;
                        RETURN;
                        END
                END
        IF P2 = NULL;
                COMPARE = TRUE;
        ELSE
                COMPARE = FALSE;
        END
```

If the encapsulation LSTACK exports this COMPARE subroutine, two stacks may be compared, as follows:

```
TABLE REQUESTS LSTACK;
TABLE BACKLOG LSTACK;

IF REQUESTS = BACKLOG;
      BACKUP;
```

UPDATING ENCAPSULATED DATA

If an encapsulated data object is given in the protection-list of an updated-statement, the encapsulation must export the names ENTRY and COMPLETION. If the encapsulation includes a definition for the subroutines ENTRY and COMPLETION, these subroutines are used to lock and unlock the data on entry and exit from the update-statement. If not, the default semantics for locking and unlocking protected data are applied.

ENTRY and COMPLETION, if defined in the encapsulation, must both be procedures with no parameters. If one is defined in the encapsulation, the other must be defined. Similarly, if one is exported, the other must be exported.

SECTION 3

SUMMARY

SUMMARY

ZONES

```
ZONE FIRST;

    BEGIN
    TABLE SPACE (1 : 12);
          ITEM ONESPACE U;
       TABLE AVAILTAB (1 : 12) T;
          ITEM AVAILABLE B;

    PROC ALLOC (ENTSIZE) P;
       BEGIN
       ITEM ENTSIZE U;
       ALLOC = NULL;
       FOR I : 1 TO 12;
          IF AVAILABLE (I);
             BEGIN
             ALLOC = LOC(ONESPACE (I));
             FOR J : 1 TO ENTSIZE;
                AVAILABLE (I - 1 + J) = FALSE;
             END
       END

    PROC DEALLOC (PTR, ENTSIZE);
       BEGIN
       ITEM PTR P;
       ITEM ENTSIZE U;
       INDEX = 1 + ((* U *)(PTR) - (LOC(SPACE)))
         / LOCSINWORD;
       FOR I : INDEX TO INDEX + ENTSIZE - 1;
          AVAILABLE (I) = TRUE;
       END

    PROC INITIALIZE
       BEGIN
       FOR I : 1 TO 12;
          BEGIN
          AVAILABLE (I) = TRUE;
          ONESPACE (I) = 0;
          END
       END
    END

TYPE LISTYPE IN FIRST TABLE
    BEGIN
    ITEM VALUE U;
    ITEM ANSWER B 4;
    ITEM NEXTOPTR LISTYPE;
    END
ITEM LISTPTR P LISTYPE;
```

storage allocated in the zone; first table used with NEW, FREE; second table to keep track of availability of space

invoked when NEW function-call appears in program; takes one input parameter (integer); return-value is an untyped pointer, implicitly converted to a typed pointer and returned as the value of NEW

invoked when FREE procedure-call appears in program; takes two input parameters (first one an untyped pointer, second one an integer)

implicitly invoked after storage allocated for the table SPACE, AVAILTAB

end of zone-declaration

declarations using the zone

Zone-Heading

The zone-heading can include information about the allocation permanence, the location in memory, and the number of words in the zone. The form of a zone-heading is:

[STATIC] [absolute-address] [W zone-size]

The zone-size in a zone-heading gives the number of words to be allocated for a zone.

Absolute-address is given with a POS-clause. An absolute address may be given if the zone has static allocation explicitly or by default.

If zone-size is given, all tables declared in the zone must be specified tables.

A zone may not be given PROTECTED allocation.

TABLE ENCAPSULATIONS

```
TABLE DATA ENCAPSULATION;
   BEGIN
   EXPORTS QUARTER'AVE, SEM'AVE, INIT,
     ASSIGN, COMPARE, READONLY COURSE
     WITH (LETTER);

   TABLE COURSE (1 : 4, 1 : 7);
      BEGIN
      ITEM COURSENAME C 7;
      ITEM HOMEWORK U 0 : 5;
      ITEM TEST U 0 : 5;
      ITEM FINAL F 0.0 : 5.0;
      ITEM LETTER C;
      END

   PROC QUARTER'AVE (QTR, HOUR);
      subroutine-definition
   PROC SEM'AVE (SEM, HOUR);
      subroutine-definition

   PROC INIT;
      subroutine-definition

   PROC ASSIGN (SOURCE : TARGET);
      subroutine-definition
   PROC COMPARE (LEFT, RIGHT) B;
      subroutine-definition
   END
```

export-list makes given names visible with read-write access unless specified otherwise; table items are exported in a with-phrase

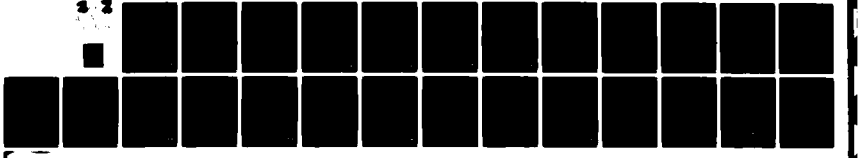
table-accessible to any subroutines in the encapsulation; not visible unless exported; all names must be fully qualified

subroutines used to manipulate encapsulated data; exported and made visible

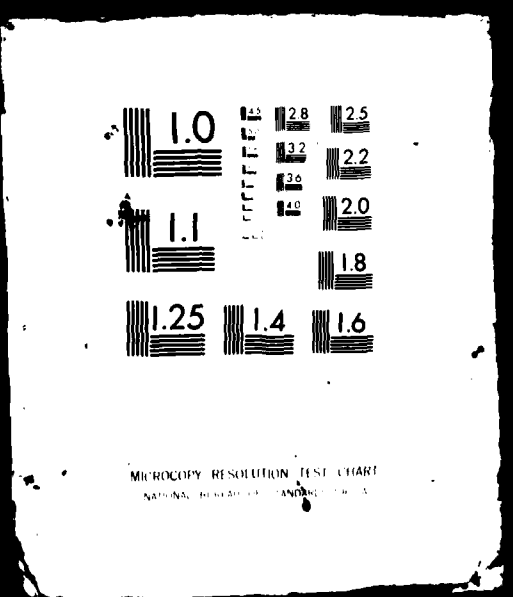
subroutine used to preset data, exported and made visible

subroutines to override default semantics of assign (=) and compare (= , <>); exported and made visible

AD-A108 529 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME 13. INTRODUCTION TO THE CHOL --ETC(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-13 NL



END
DATE FILMED
1 82
DTIC



TYPE ENCAPSULATIONS

```
TYPE REPORT'CARD ENCAPSULATION;
   BEGIN
   EXPORTS QUARTER'AVE, SEM'AVE, INIT,
     ASSIGN, COMPARE, READONLY COURSE
     WITH (LETTER);

   TABLE COURSE (1 : 4, 1 : 7);
      BEGIN
      ITEM COURSENAME C 7;
      ITEM HOMEWORK U 0 : 5;
      ITEM QUIZ U 0 : 5;
      ITEM TEST U 0 : 5;
      ITEM FINAL F 0.0 : 5.0;
      ITEM LETTER C
      END

   PROC QUARTER'AVE (REPORTAB, QTR, HOUR);
      subroutine-definition
   PROC SEM'AVE (REPORTAB, SEM, HOUR);
      subroutine

   PROC INIT (REPORTAB);
      subroutine-definition

   PROC ASSIGN (SOURCE : TARGET);
      subroutine-definition
   PROC COMPARE (LEFT, RIGHT) B;
      subroutine-definition
   END
```

export-list makes given names visible with read-write access unless specified otherwise; table items are exported in a with-phrase

table accessible to any subroutines in the encapsulation; not visible unless exported; all names must be fully qualified

subroutines used to manipulate encapsulated data; exported and made visible; REPORTAB will be declared to be of type REPORT'CARD; *the actual parameter* bound to REPORTAB may be any table of type REPORT'CARD

subroutines to override default semantics of assign (=) and compare (- , <>); exported and made visible

THE JOVIAL (J73) WORKBOOK

VOLUME 15

CONDITION HANDLING

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 15 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape. Specifically addressed language features include built-in and programmer defined conditions, the signal-statement, condition handlers and the !SUPPRESS directive.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction- a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

CONDITION HANDLING

SOFTECH

CONDITION HANDLING

A condition is an exceptional event that arises during program execution. Some conditions indicate errors and other conditions indicate special events. Two kinds of condition are permitted in JOVIAL (J73/C), built-in-conditions and programmer-defined-conditions.

The condition handling capability of JOVIAL (J73/C) provides for the interruption of the normal control sequence should an exceptional event occur.

BUILD-IN CONDITIONS

Built-in-conditions are conditions that are predefined in JOVIAL (J73/C). Most of these conditions are associated with language violations that occur at run-time. Many language violations are detected and reported by the compiler at compile-time; some violations may only be detected when the program is running. When such a language violation occurs, the condition associated with the violation is signalled by the system.

The built-in-conditions and the reasons for signalling each condition are given in the following list:

Condition	Reason for Signalling
OVERFLOW	The value computed in a numeric formula is not within the range of values that may be accommodated in a formula of that size.
ZERODIVIDE	The divisor of a formula has the value 0.
RANGEERROR	A numeric value is assigned to an item that is either outside the specified or default value-range for the item.

Condition	Reason for Signalling
CASEERROR	The case-selector of a case-statement has a value that does not match any case-index, and the case-statement does not have a default-option.
VARIANTERROR	A name declared in one variant is referenced when the variant tag has a value that indicates that another variant is present.
SUBSCRIPTRANGE	A subscript is given that is outside the range declared for the dimensioned table.
STRINGRANGE	A bit or character is selected that is outside the length of a bit or character string.
POINTERDEREFERENCE	The pointer used in the dereference has the value NULL.
CONVERSION	An explicit conversion is applied to a formula that has a value that may not be accommodated in the type given by the conversion.
STORAGE	The NEW function is called and the necessary amount of storage is not available from heap storage.
NEXTERROR	The NEXT function is called with a parameter that indicates a next or previous status-constant that is outside the set of status-constants.
ABORT	An abort-statement is executed or the ABORT condition is signalled. The ABORT condition is the only built-in condition that occurs only as a result of some programmer action.

PROGRAMMER-DEFINED CONDITIONS

A programmer may declare a condition in a condition-declaration. The form is:

CONDITION condition-name , ... ;

Example

CONDITION NEARFAIL;

CONDITION NEARFAIL, APLUS;

A condition-declaration may appear anywhere a declaration may be given.

THE SIGNAL-STATEMENT

The signal-statement is used to signal the occurrence of a condition. Programmer-defined-conditions occur only if signalled in this way. Build-in conditions, with the exception of the ABORT condition, are automatically signalled by the system when a language violation occurs. Built-in-conditions, may also be signalled by the signal-statement.

The form of the signal-statement is:

SIGNAL condition ;

A signal-statement indicates an occurrence of the specified condition. For example:

SIGNAL REDALERT;

This statement signals the occurrence of the condition REDALERT.

CONDITION HANDLERS

A condition occurs either by an implicit signal from the system, if a language violation occurs, or by an explicit signal in a signal-statement. Once a condition occurs, it must be resolved. The mechanism for resolving a condition is called a condition-handler. A condition-handler provides actions to be taken if a specified set of conditions arise.

The form of a condition-handler is:

```
HANDLER

        [ ( DEFAULT ) : statement ]

        [ ( condition ,... ) : statement ]
```

A handler may have either a default option, a sequence of one or more specified options, or both, as indicated by square brackets. A handler must have at least one option.

The default option, if present, provides the action to be taken for any condition that is not provided for explicitly.

For example, given the following handler:

```
HANDLER

        (DEFAULT): RECOVERY;
```

This handler contains only a default option. It calls procedure RECOVERY if any condition occurs.

The specified option provides actions for particular conditions. A condition may be either a programmer-defined condition or a bulit-in-condition.

For example:

```
HANDLER

        (DEFAULT):                          RECOVERY

        (ZERODIVIDE):                       ZERODIV = ZERODIV + 1;

        (OVERFLOW, FLOATOVERFLOW):          ANALYSIS;

        (REDALERT)                          COMMANDCONTROL;
```

This handler increments the counter ZERODIV if the built-in-condition ZERODIVIDE occurs, calls procedure ANALYSIS if either the built-in-condition OVERFLOW or the programmer-defined-condition FLOATOVERFLOW occurs, calls procedure COMMANDCONTROL if the programmer-defined-condition REDALERT occurs, and calls procedure RECOVERY if any other condition occurs.

Program Units

A condition-handler is associated with a program unit. A program unit is either a compound-statement, a subroutine-body, or a program-body. The handler is given at the end of a program unit, just before the terminating END.

The form of a compound-statement with a condition-handler is:

BEGIN

statement ...

[condition-handler]

[label ...] END

The form of a subroutine-body or program-body with a condition-handler is:

BEGIN

[declaration ...]

statement ...

[subroutine-definition ...]

[condition-handler]

[label ...] END

Control passes through a condition-handler only when a condition occurs.

Activation of a Condition Handler

When a condition occurs in a program unit, search for a handler begins. If the condition-handler in the current program unit contains an option for that condition or a default option, that option is executed. After it is executed, the program unit that includes the handler is terminated. If the execution of the option in the condition-handler did not transfer control, control returns to point following the invocation of the program unit.

If the program unit does not have a condition-handler or if the appropriate option is not present in the condition-handler and the condition-handler does not have a default option, execution of the program unit is terminated and the condition is propagated out of the program unit. That is, the program unit that invoked the current program unit is examined to see if it can handle the signalled condition. The program unit that invoked a program unit is called its dynamic predecessor. If the dynamic predecessor contains either an option for the condition or a default option, the handler option is executed and that program unit terminated. If it does not, the dynamic predecessor's program unit is terminated and its dynamic predecessor examined.

This process continues until a handler is found for the condition or until the program is terminated.

Examples

1) The following example defines a subroutine in which the STRINGRANGE condition may occur. Subroutine GETCHAR finds the first non-blank character in a string (STRING), starting from a given position (INDEX).

```
PROC GETCHAR (STRING, INDEX) C;
        BEGIN
        ITEM STRINC C *;
        ITEM INDEX U;
        FOR INDEX : INDEX BY 1 WHILE BYTE (STRING, INDEX, 1) =
        ' ';
              BEGIN
              END
        GET CHAR = BYTE (STRING, INDEX, 1);
        END
```

Given the following item-declaration:

```
ITEM MESSAGE C 40;
```

The following function-call produces a STRINGRANGE condition:

```
CHAR = GETCHAR  (MESSAGE, 42);
```

Any value of INDEX greater than or equal to 40 used in a call to GETCHAR for the string MESSAGE produces a STRINGRANGE condition.

A STRINGRANGE condition may occur if the input string ends with a string of blanks. In that case, subroutine GETCHAR is unable to locate a non-blank character within the limits of the string and runs off the end of the string.

If every valid string given to GETCHAR is set up to include an end-of-message character at the end, the STRINGRANGE condition occurs only in case of an error.

Subroutine GETCHAR does not have a condition-handler, so if the STRINGRANGE condition occurs, the GETCHAR function is terminated abnormally and the search begins for a condition-handler.

2) In the following example, GETCHAR is invoked in subroutine FINDCHAR, which counts the occurrences of a particular character in a string:

```
PROC FINDCHAR (STRING, CHAR : COUNT);
    BEGIN
    ITEM STRING C *;
    ITEM CHAR C;
    ITEM COUNT U;
    COUNT = 0;
    FOR INDEX : 0 WHILE INDEX <= BYTESIZE (STRING);
        IF GETCHAR (STRING, INDEX) = CHAR;
            COUNT = COUNT + 1;
    PROC GETCHAR (STRING, INDEX) C;
        BEGIN
        ITEM STRING C *;
        ITEM INDEX U;
        FOR INDEX : INDEX BY 1 WHILE BYTE
        (STRING, INDEX, 1) = ' ';
            BEGIN
            END
        GETCHAR = BYTE (STRING, INDEX, 1);
        END
    HANDLER
        (DEFAULT): ID = 10;
    END
```

If the string input to FINDCHAR does not have an end-of-message character at the end, the STRINGRANGE condition may be signalled. GETCHAR does not have a handler, so the condition is propagated out to FINDCHAR, which has a handler. This handler has a default option, so it handles the condition STRINGRANGE. ID, some global variable, is set to 10 and FINDCHAR terminated. Control returns to the point after subroutine FINDCHAR was called.

3)

```
BEGIN "OUTER"
...
IF QUARTER = 4;
   BEGIN "MIDDLE"
   ...
   FOR I : 1 TO 30;
      BEGIN "INNER"
      ...
      IF INDEX > 42;
         SIGNAL SUBSCRIPTRANGE;
      IF INDEX - I = 0;
         SIGNAL ZERODIVIDE;
      IF GRADE = 'A +';
         SIGNAL APLUS;
      ...
      HANDLER
         (APLUS, NEARFAIL) : ...
      END
   ...
   HANDLER
      (ZERODIVIDE) : ...
   END
...
HANDLER
   (SUBSCRIPTRANGE) : ...
END
```

In this example, if APLUS is signalled in the inner program unit, the APLUS handler is executed and the inner program unit is terminated.

If ZERODIVIDE is signalled in the inner program unit, the appropriate handler is not found there, that program unit is terminated, and the condition is propagated out to the middle program unit. The middle program unit is examined for the appropriate handler; it is found; it is executed; and the middle program unit is terminated.

If SUBSCRIPTRANGE is signalled in the inner program unit, the appropriate handler is not found there, that program unit is terminated and the condition is propagated out to the middle program unit. The middle program unit is examined for the appropriate handler, one is not found, that program unit is also terminated and the condition is propagated out to the outer program unit. The outer program unit is examined for the appropriate handler; it is found; it is executed; and the outer program unit is terminated.

SIGNALLING WITHIN HANDLERS

The signal-statement may be used within a handler to indicate the occurrence of a condition.

For example, a program unit may have a set of handlers; each one does a certain amount of specific processing and signals a general handler, to complete the condition processing.

For instance, subroutine FINDCHAR may have a STRINGRANGE option in the handler, which sets ID to 10 and signals the programmer-defined-condition BADSTRING.

```
CONDITION BADSTRING;
PROC FINDCHAR (STRING, CHAR : COUNT);
        BEGIN
        ITEM STRING C *;
        ITEM CHAR C;
        ITEM COUNT U;
        COUNT = 0;
        FOR INDEX : 0 WHILE INDEX <= BYTESIZE (STRING);
                IF GETCHAR (STRING, INDEX) = CHAR;
                        COUNT = COUNT + 1;
        PROC GETCHAR (STRING, INDEX) C;
                BEGIN
                ITEM STRING C *;
                ITEM INDEX U;
                FOR INDEX : INDEX BY 1 WHILE BYTE
                        (STRING, INDEX, 1) = ' ';
                        BEGIN
                        END
                GETCHAR = BYTE (STRING, INDEX, 1);
                END
        HANDLER
                (DEFAULT:       ID-10;
                (STRINGRANGE): BEGIN
                                ID=10
                                SIGNAL BADSTRING;
                                END
END
```

If the condition STRINGRANGE occurs when FINDCHAR is being executed, it is handled by the handler option for STRINGRANGE associated with that subroutine. ID is set to 10 and to programmer-defined-condition BADSTRING is signalled. FINDCHAR is terminated abnormally and the search for a handler for the BADSTRING condition begins.

SIGNALLING THE RECURRENCE OF A CONDITION

The signal-statement may be used to indicate the recurrence of a condition. It is a signal-statement without a condition name, as follows:

```
SIGNAL;
```

This form of the signal-statement is used only within a handler. It indicates the recurrence of the condition that caused the initial execution of the handler. In this way, some local processing may be done before the condition is propagated to an outer level for more extensive handling.

In the FINDCHAR example, GETCHAR may have a handler for STRINGRANGE that sets a flag and propagates out the STRINGRANGE condition.

```
CONDITION BADSTRING;
PROC FINDCHAR (STRING, CHAR : COUNT);
    BEGIN
    ITEM STRINC C *;
    ITEM CHAR C;
    ITEM COUNT U;
    COUNT = 0;
    FOR INDEX : 0 WHILE INDEX <= BYTESIZE (STRING);
        IF GETCHAR (STRING, INDEX) = CHAR;
            COUNT = COUNT + 1;
```

```
PROC GETCHAR (STRING, INDEX) C;
        BEGIN
        ITEM STRINC C *;
        ITEM INDEX U;
        FOR INDEX : INDEX BY 1 WHILE BYTE (STRING,
        INDEX, 1) = ' ';
                GETCHAR = BYTE (STRING, INDEX, 1);
        HANDLER
                (DEFAULT): BEGIN
                           CHARFLAG = FALSE;
                           SIGNAL;
                           END
        END
HANDLER
        (DEFAULT):       ID = 10;
        (STRINGRANGE): BEGIN
                       ID = 10;
                       SIGNAL BADSTRING;
                       END
        END
```

If the STRINGRANGE condition occurs within GETCHAR, it is handled by the handler in GETCHAR, which sets CHARFLAG to FALSE and signals a reoccurrence of the STRINGRANGE condition.

The STRINGRANGE condition signalled in the GETCHAR handler is propagated out and handled by the STRINGRANGE option in the FINDCHAR handler, which sets ID to 10 and signals the BADSTRING condition. The BADSTRING condition is handled by the first explicit handler for that condition or default handler that is encountered in searching back through the dynamic predecessors of FINDCHAR.

HANDLERS WITHIN HANDLERS

A handler may include a nested handler. This nested handler is invoked to handle any conditions that occur while the handler is processing a signal.

For example:

```
PROC BUILDCHART;
    BEGIN
    ITEM TALLY U 10, 0 : 1000;
    ...
    HANDLER
        (ZERODIVIDE): BEGIN
                      TALLY = TALLY + 1;
                      HANDLER
                          (RANGEERROR): ERRTALLY = TRUE;
                      END
END
```

If the ZERODIVIDE condition occurs in subroutine BUILDCHART, it is processed by the handler in BUILDCHART. During the processing of the ZERODIVIDE condition, if the RANGEERROR condition occurs, it is processed by the handler nested within the ZERODIVIDE handler. When the RANGEERROR condition handling is complete, the ZERODIVIDE handler is terminated. When the ZERODIVIDE condition handling is complete, BUILDCHART is terminated.

SECTION 2

SUPPRESSING CONDITIONS

THE !SUPPRESS DIRECTIVE

The !SUPPRESS directive is used to direct the compiler to omit generating code to check for the existence of the named conditions. This directive has the form:

!SUPPRESS condition ,... ;

One or more conditions, separated by commas, may be given. The !SUPPRESS directive must appear before the first executable statement in a program unit. The !SUPPRESS directive applies to the immediate program units and nested program units, but not to nested subroutines. Within the affected program units, the occurrence of the condition named in a !SUPPRESS directive is ignored.

The !SUPPRESS directive may be included in a program to save time and space after the program has been thoroughly checked and is ready to be placed in operation.

If a condition designated in a !SUPPRESS directive is a predefined condition, the !SUPPRESS directive is not an assertion that the condition will not arise. The condition may indeed arise during program execution, but a handler for the condition is not executed. As a result, the program is invalid and its behavior unpredictable. If a !SUPPRESS directive specifies a programmer-defined-condition, a signal-statement for that condition has no effect.

SOFTECH, INC.
Publications Remarks Form

TITLE: The JOVIAL (J73) Workbooks, Volumes 1 - 15

ORDER NO. 1081-1

DATED April 1981

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE -
NOTE: U.S. Postal Service will not deliver stapled forms.

PLEASE PLACE STAMP HERE

SOFTECH, INC.

460 Totten Pond Road

Waltham, MA 02154

ATTN: Quality Assurance

SOFTECH

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.